TKK Dissertations 202
Espoo 2009

XML-AWARE DATA SYNCHRONIZATION FOR MOBILE DEVICES

Doctoral Dissertation

Tancred Lindholm

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

TKK Dissertations 202
Espoo 2009

XML-AWARE DATA SYNCHRONIZATION FOR MOBILE DEVICES

Doctoral Dissertation

Tancred Lindholm

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences for public examination and debate in Auditorium T1 at Helsinki University of Technology (Espoo, Finland) on the 11th of December, 2009, at 12 noon.

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering
P.O. Box 5400
FI - 02015 TKK
FINLAND
URL: http://www.cse.tkk.fi/
Tel. +358-9-47001
E-mail: tancred.lindholm@hiit.fi

ISBN 978-952-248-212-9
ISBN 978-952-248-213-6 (PDF)
ISSN 1795-2239
ISSN 1795-4584 (PDF)
URL: http://lib.tkk.fi/Diss/2009/isbn9789522482136/

TKK-DISS-2685

Picaset Oy
Helsinki 2009

ABSTRACT OF DOCTORAL DISSERTATION	HELSINKI UNIVERSITY OF TECHNOLOGY P. O. BOX 1000, FI-02015 TKK http://www.tkk.fi

Author Tancred Lindholm

Name of the dissertation

XML-aware Data Synchronization for Mobile Devices

Manuscript submitted May 26, 2009	Manuscript revised November 6, 2009
Date of the defence December 11, 2009	
☐ Monograph	☒ Article dissertation (summary + original articles)
Faculty	Faculty of Information and Natural Sciences
Department	Department of Computer Science and Engineering
Field of research	Data Synchronization and Mobile Computing
Opponent(s)	Professor Liviu Iftode
Supervisor	Professor Antti Ylä-Jääski
Instructor	Professor Kimmo Raatikainen

Abstract

In everyday life, and when using computer systems in particular, it is sometimes the case that a logical datum is replicated into multiple copies, such as when we send a document by electronic mail, or inform interested parties of a new address of residence. If the datum for some reason changes, we would then also like the changes to be reflected in the copies. The problem of keeping the copies up-to-date with respect to each other is studied under the heading of data synchronization.

In this thesis, we address data synchronization for mobile devices with limited energy resources and limited connectivity to the Internet, such as mobile phones. The importance of data synchronization is emphasized here, as it becomes infeasible to communicate continuously and in high volumes about the current state of each copy. The established conventions of the Internet and mobile computing environments on such matters as storage interfaces and data formats define an overall system architecture, into which we as seamlessly as possible want to incorporate our proposal. By focusing on interoperability we lower the threshold for utilizing our research in practice.

We present a comprehensive approach to data synchronization for mobile devices that is optimistic and state-based, and which targets opaque and XML files on a standard file system. We consider how to use the available connectivity in an economical manner, and so that existing sources of data on the Internet can be utilized. We focus on XML synchronization, where we identify an opportunity to utilize the structure of the data the format exposes. Specifically, we present an algorithm for merging concurrent changes to XML documents which supports subtree moves, an efficient heuristic algorithm for computing tree-level changes between two XML documents, and an overall architecture and algorithms to support the use of lazily instantiated XML documents. Our data synchronization approach is evaluated quantitatively in several experiments, as well as qualitatively by constructing applications that build on top of the approach. One of our applications is an editor that processes 1 GB XML files on a mobile phone.

Keywords	mobile computing, data synchronization, XML		
ISBN (printed)	978-952-248-212-9	ISSN (printed)	1795-2239
ISBN (pdf)	978-952-248-213-6	ISSN (pdf)	1795-4584
Language	English	Number of pages	80 + app. 64
Publisher	Department of Computer Science and Engineering, Helsinki University of Technology		
Print distribution	Department of Computer Science and Engineering		
☒ The dissertation can be read at http://lib.tkk.fi/Diss/2009/isbn9789522482136/			

SAMMANFATTNING (ABSTRAKT) AV DOKTORSAVHANDLING	TEKNISKA HÖGSKOLAN PB 1000, FI-02015 TKK http://www.tkk.fi

Författare Tancred Lindholm

Titel

Datasynkronisering med XML-stöd för mobila fickdatorer

Inlämningsdatum för manuskript den 26e maj, 2009	Datum för disputation den 11e december, 2009
Datum för det korrigerade manuskriptet den 6e november, 2009	
☐ Monografi	☒ Sammanläggningsavhandling (sammandrag + separata publikationer)
Fakultet	Fakulteten för informations- och naturvetenskaper
Institution	Institutionen för datateknik
Forskningsområde	Datasynkronisering, mobil IT
Opponent(er)	Professor Liviu Iftode
Övervakare	Professor Antti Ylä-Jääski
Handledare	Professor Kimmo Raatikainen

Sammanfattning (Abstrakt)

Vi stöter dagligen, och speciellt när vi använder IT, på situationer då information mångfaldigats så att kopior uppstår. Detta sker exempelvis då vi använder e-post eller meddelar ny adress efter att ha flyttat. Om informationen ändras vill vi att ändringarna syns även i kopiorna. Inom datasynkronisering studeras hur vi kan hålla kopior av information aktuella.

I denna avhandling behandlar vi datasynkronisering för bärbara datorer med begränsad tillgång till energi och bredband, som exempelvis mobiltelefoner. Datasynkronisering är speciellt viktig i denna miljö, eftersom den inte lämpar sig för kontinuerlig datakommunikation eller för att överföra stora mängder data i syfte att hålla alla kopior aktuella i det ovannämnda fallet. Etablerade konventioner på internet och inom mobil IT gällande t.ex. gränssnitt och format för lagring av data definierar en övergripande systemarkitektur i vilken vi vill infoga vårt bidrag så friktionsfritt som möjligt. Genom att fokusera på kompatibilitet underlättar vi användningen av vår forskning i praktiska tillämpningar.

Vi presenterar en komplett metod för synkronisering av data på mobila fickdatorer som är optimistiskt och s.k. *state-based*. Metoden lämpar sig för synkronisering av filer i ett filsystem, speciellt sådana som innehållar data i XML-formatet. Vi överväger hur man kan använda det tillgängliga datanätet sparsamt, och på ett sätt som möjliggör användning av existerande datakällor på internet. Vi granskar synkronisering av XML, och identifierar möjligheter i den exponering av struktur som XML medför. Specifikt presenterar vi en algoritm för att införliva (*merge*) jämsides löpande ändringar i XML-dokument som stöder flyttning av trädstrukturer, en effektiv heuristisk algoritm för att beräkna ändringar i trädstrukturen hos ett XML-dokument, och en övergripande arkitektur med tillhörande algoritmer som möjliggör att XML-dokument laddas i arbetsminnet efter behov. Vi utvärderar synkroniseringsmetoden kvantitativt i ett antal experiment, samt även kvalitativt genom att konstruera applikationer som bygger på våra resultat. Bland de applikationer vi konstruerat finns ett redigeringsprogram som kan hantera 1 GB stora XML-filer i en mobiltelefon.

Ämnesord (Nyckelord)	mobil IT, datasynkronisering, XML		
ISBN (tryckt)	978-952-248-212-9	ISSN (tryckt)	1795-2239
ISBN (pdf)	978-952-248-213-6	ISSN (pdf)	1795-4584
Språk	Engelska	Sidantal	80 + app. 64
Utgivare	Institutionen för datateknik, Tekniska Högskolan		
Distribution av tryckt avhandling	Institutionen för datateknik		

☒ Avhandlingen är tillgänglig på nätet http://lib.tkk.fi/Diss/2009/isbn9789522482136/

Acknowledgments

If it were not for the inspiration, interest, and thought-provoking discussions by which others have enriched my career as a researcher this thesis would never have progressed beyond being a set of undeveloped philosophical musings of mine. I am thus most grateful towards each and every person who has taken an interest in this work, be it brief or lasting throughout the process.

The majority of the work presented here has been carried out at the Helsinki Institute for Information Technology (HIIT), at the time led by Prof. Martti Mäntylä, whom I thank, along with the Institute and its past and current personnel, for providing a stimulating environment where scientific research truly comes first and foremost. I am indebted to my instructor, Prof. Kimmo Raatikainen, who always furthered the interests of his students, and regret that he is no longer among us.

It is most fascinating to work with talented people on a daily basis, and I feel this was indeed the case in the Fuego Core project, my home at HIIT. I thank my fellow researchers at HIIT, and those who worked with me in the Fuego Core project in particular. I thank Dr. Jaakko Kangasharju for his invaluable insights and solid advice on writing, and Prof. Sasu Tarkoma for his strong support during the last two years. The funding provided by partners in industry and the National Technology Agency of Finland (TEKES) is gratefully acknowledged.

I thank the pre-examiners of this thesis, Prof. Jukka Riekki (University of Oulu) and Associate Professor Anthony D. Joseph (University of California, Berkeley) for their valuable comments and suggestions, based on which I have been able to make this thesis a better one. I would also like to thank my supervisor, Prof. Antti Ylä-Jääski, especially for helping me cut through all the red tape that goes with this process.

I would further like to thank the Google Docs team for giving me the opportunity to work on data synchronization in a large-scale distributed application in the "real world". Working where theory meets practice has been most inspiring.

I express my deepest gratitude and love to my wife Anne for her unending belief that this project would be finished, for her inspiring words to keep going forward, and for single-handedly maintaining the household during periods of intense writing. Finally, I thank my parents Mikael and Christel, my brothers, relatives, and friends.

Helsinki, November 6, 2009
Tancred Lindholm

Anschauung und Begriffe machen also die Elemente
aller unserer Erkenntnis aus, so daß weder Begriffe,
ohne ihnen auf einige Art korrespondierende Anschauung,
noch Anschauung ohne Begriffe,
ein Erkenntnis abgeben kann.

— Immanuel Kant, Kritik der reinen Vernunft
1781

Contents

Original Publications

I Tancred Lindholm. A three-way merge for XML documents. In Ethan V. Munson and Jean-Yves Vion-Dury, editors, *ACM Symposium on Document Engineering*, pages 1–10. ACM Press, October 2004.

II Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. A hybrid approach to optimistic file system directory tree synchronization. In Vijay Kumar, Arkady B. Zaslavsky, Ugur Çetintemel, and Alexandros Labrinidis, editors, *Fourth International ACM Workshop on Data Engineering for Wireless and Mobile Access*, June 2005.

III Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Fast and simple XML tree differencing by sequence alignment. In David F. Brailsford, editor, *ACM Symposium on Document Engineering*, pages 75–84, October 2006.

IV Tancred Lindholm and Jaakko Kangasharju. How to edit gigabyte XML files on a mobile phone with XAS, RefTrees, and RAXS. In *Fifth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2008)*, Dublin, Ireland, July 2008.

V Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Syxaw: Data synchronization middleware for the mobile web. *Mobile Networks and Applications*, 14(5):661–676, 2009

The author of this thesis is the main author of the above publications. In Articles II–V the other authors (Jaakko Kangasharju and Sasu Tarkoma) have contributed valuable suggestions for improvement of the subject matter and suggestions and corrections to the presentation. The XAS component mentioned in Article IV is mainly the work of Jaakko Kangasharju. The thesis author has however participated in the implementation of the XAS random access and streaming parsing facilities that are described in the Article.

Chapter 1

Introduction

My wife and I recently moved to a new location. One of the many things to remember is to make sure that everyone gets the new address, so your mail will not be appearing in the mailbox of whoever moves in at the old address. At least here in Finland, changing your address typically involves notifying several parties, and you are still likely to lose the occasional piece of mail. The root of the problem is that the address is stored in the computer systems of companies, government agencies, and contact books, both hand-written and electronic. When you move, all of these need to be updated, and in reality, they seldom are.

The example above is a typical instance of a data synchronization problem. Namely, there is a single logical datum, in this case the current address of my wife and I, of which there exists several copies. When our address changes so should each copy.

Readers familiar with a modern office setting where electronic correspondence is common are likely to have encountered a variation of this problem when collaborating on some document. Copies of the document are edited by co-workers, and are passed around by means of electronic mail and portable storage ("USB sticks" are popular at the moment). However, in the end, a single revision incorporating the edits of the collaborators is needed, and producing this revision by carefully surveying copies for changes can be quite tedious.

Data synchronization has been researched for several decades. Much progress has been made, both in terms of theory and on practical aspects. It would, however, be premature to say that data synchronization is "solved". The examples we gave above happen every day. Data synchronization is an area where improvements and inventions are still needed, stated Bill Gates in his keynote at the 2008 Consumer Electronics Show [31].

Data synchronization is not yet as effortless and automated as one could envision. We believe this is largely due to challenges in the applied aspects of data synchronization, rather than due to weaknesses in the theory. A key

issue here appears to be accommodating the large amount of existing software systems and data sources that could benefit from synchronization. It therefore seems worth focusing on providing synchronization in a manner that is *interoperable* with existing systems.

Consequently, in this thesis, data synchronization is treated from an applied research perspective with the principle of interoperability with existing systems as an overall guideline.

1.1 Research Question, Scope, and Objectives

Consider a *distributed computing* environment [20] where a number of computational *nodes* exchange data by passing *messages* between each other. Furthermore, in this particular case, each node harbors a data object that may be subjected to *edits* by an entity local to the node, such as an application or an end user.

In this setting, we can formulate the theoretical *data synchronization problem* considered in this thesis as follows: how can we make the data objects consistent by passing messages about the edits between the nodes? The answer can be stated in the form of a *synchronizer*, which is an algorithm that implements a solution to the synchronization problem.

The practical research aspect rises from the environment in which we consider the theoretical synchronization problem. We address data synchronization for a mobile environment consisting of devices with limited energy resources and limited connectivity to the Internet, such as mobile phones. The importance of data synchronization is emphasized here, as it becomes infeasible to communicate continuously and in high volumes about the current state of each copied object. The established conventions of the Internet and mobile computing environment on such matters as storage interfaces and data formats define an overall system architecture, into which we as seamlessly as possible want to incorporate our proposal.

When several copies of a data object get modified in isolation the synchronizer needs to resolve how the modifications should be combined. This often requires knowledge of the structure of the data object, which has traditionally not been available, as closed and proprietary storage formats have been common. However, more recently, the benefits of an open structure have been recognized, and have even received some attention in mainstream media (e.g. [79]). Currently, the main vehicle for open structure is the Extensible Markup Language (XML) [143]. A synchronizer that is able to leverage XML structure when synchronizing XML data, for instance in the process of propagating or applying edits, or to successfully combine modifications, is said to be *XML-aware*.

Besides using a widely supported data format, being interoperable with existing applications means supporting a storage interface that is widely

used. The standard hierarchical file system interface (e.g., [119]) is ubiquitously supported, and thus becomes the natural choice.

This setting motivates the research question which we investigate in this thesis:

> *How can we provide XML-aware file synchronization in a manner that is compatible with existing applications and suited for the mobile environment?*

The first objective of the research then becomes the construction of a data synchronizer suited for mobile devices and which synchronizes file systems and XML files in particular. The second objective is to evaluate the proposed synchronizer for utility and fitness to the mobile environment.

1.2 Structure of the Thesis

To define more precisely what is meant by the "mobile environment", we start by presenting the mobile computing environment considered here and its implications on software design in Chapter 2. We then move on to an overview of the field of data synchronization in Chapter 3, in order to put our work in context. We present the theoretical foundations of synchronization and the area of optimistic data synchronization, while focusing on the so-called state-based approach, and in particular its application to file system synchronization, according to the scope of the research carried out in this thesis. XML-specific aspects of synchronization are considered in Chapter 4, with XML differencing and merging forming the topics.

Against the background of the target environment and data synchronization, we are in a position to consider the design of data synchronizers in the mobile environment, and to further motivate the chosen scope of state-based optimistic synchronization. We do this in Chapter 5 where we also review the design of synchronizers that have been constructed for the mobile environment.

In Chapter 6 we present the contributions of this thesis, which are organized into the topics of XML merging and differencing, efficient use of XML, and the construction of an XML-aware data synchronizer for the mobile environment, which builds on the previous topics. The thesis is concluded with a discussion in Chapter 7.

1.3 Summary of Contribution

The contribution of this thesis consists of this Introduction and Articles I–V, whose main contributions are as follows.

Article I contributes the design, implementation, and evaluation of a three-way merge for XML with support for subtree moves.

Article II contributes the design, implementation, and evaluation of three-way merging and other algorithms for partially instantiated XML documents, and a method for efficiently scanning a file system for changes.

Article III contributes the design, implementation, and evaluation of a high-performance XML differencing algorithm, and a quantitative comparison with related work.

Article IV contributes an XML processing system based on partially instantiated XML documents that provides functionality for document mutability, data synchronization, and document versioning. The system is implemented and evaluated, demonstrating that it enables processing of XML files in the order of 1 GB on mobile phones.

Article V contributes the design, implementation, and evaluation of an XML-aware data synchronizer suited for interoperable synchronization on mobile devices. The evaluation consists of quantitative measurements of the techniques used to improve network utilization, and a study of the suitability of the synchronizer for providing synchronization services on a mobile device.

The complete set of algorithms and other software components presented in the above Articles are available as Open Source to facilitate dissemination and verification of the results, as well as to lower the threshold for transfer of the results to practitioners and industry. The detailed contributions with pointers to source code are given in Chapter 6.

1.4 Research Methodology and History

The research presented in this thesis was carried out in the Fuego Core project[1] series (2002–2007) at the Helsinki Institute for Information Technology[2]. The aim of the project was to develop a set of middleware services for mobile devices, and the overall vision of the project was to emphasize the qualities of interoperability, openness, and computational simplicity.

The project was divided into 1-year cycles to support an iterative, incremental, and constructive approach to research. Each cycle, as applicable to the work carried out here, consisted of the following steps:

[1] `http://www.hiit.fi/fuego/fc`
[2] `http://www.hiit.fi`

1. Analyze previous work outside the project and the results of previous research cycles
2. Identify a research topic where progress over the current state seems possible
3. Establish a set of requirements for an improved approach
4. Propose a specification for an improved approach
5. Implement the improved approach as computer software
6. Evaluate the approach and implementation with a focus on quantitative measurements and the requirements established in Step 3.

In each cycle, the overall aim of constructing a data synchronizer for mobile devices guided the choice of relevant research topics. The construction of software in each cycle provided concrete systems that could be tested against acceptance criteria, such as memory footprint, network usage patterns, etc. In particular, having testable systems meant that infeasible approaches could be rejected early on.

During 2002–2003 a prototype data synchronizer, "XMLFS", was built, which, however, did not execute on authentic devices. Still, realistic network usage could be observed with the aid of a cellular network interface. With XMLFS we proposed a layered file system approach, rather than a file system synchronizer. We identified merging of XML documents as a viable research topic, which resulted in the publications [61] and Article I.

In 2004, we abandoned the idea of a file system in favor of a file synchronizer. This was motivated by the identification of topics specific to XML synchronization for the next research steps, and by the requirement to move towards a more platform-neutral design. The goal to use XML throughout the system as far as possible lead us to investigate synchronization of file system directory trees expressed as XML.

As a heritage from the file system approach we still considered synchronization of complete file systems with thousands of entries, rather than a limited subset. In doing this we encountered scalability issues with a state-based approach to directory tree synchronization. However, as the state-based approach had several advantages, we did not abandon it in favor of an edit-based approach, but rather identified efficient state-based directory tree synchronization as a research topic. The results were published in Article II.

In 2005 we considered how the synchronization model could be extended with more advanced structures for managing causality between changes. The goal was to allow direct device-to-device synchronization in cases when the network infrastructure was temporarily unavailable. We also considered partial synchronization of XML data by attaching queries

to the synchronization control metadata. Although functional, this work was in the end not carried out further, as we pre-empted it with work on efficient XML synchronization which seemed more viable. That work was the construction of the XML differencing algorithm presented in Article III.

An important development during 2005 and especially 2006 was that the data synchronizer, now named Syxaw, was ported to a mobile device, the Nokia 9500 Communicator smartphone. This enabled us to construct sample applications (e.g., [50]) and to obtain authentic measurements.

During 2006 and 2007, we focused on elaborating ideas originating in the state-based directory tree synchronization work in Article II. An integrated approach for efficient read and write random access to verbatim XML documents was developed, and published in Article IV.

During 2008 we elaborated on the interoperability aspects of the synchronization protocol used by Syxaw. The Syxaw synchronizer was subsequently published in Article V.

Chapter 2

The Mobile Computing Environment

The research in this thesis targets an environment of personal communications devices with support for Internet networking, such as mid- to high-end mobile phones. These devices are commonly referred to as *smartphones*. The setting belongs to the domain of *mobile computing* [105], and its descendant fields of *pervasive* and *ubiquitous* computing [106, 132]. While the Internet has traditionally been the domain of non-mobile nodes, mobile Internet usage is on the rise, and has been envisioned to outgrow fixed-node networking eventually [56].

A representative smartphone at the time of writing shown in Figure 2.1 is the "Dream" from HTC Corporation[1]. The phone has a built-in camera, a Global Positioning System (GPS) [40] receiver along with other sensors, and supports cellular, Wireless LAN (WLAN) [44], and Bluetooth [10] wire-

Figure 2.1: The "Dream" smartphone from HTC Corporation

[1] http://www.htc.com/

less access technologies. The phone runs the Android operating system[1]. Applications include a browser for the Word Wide Web, Email and Calendar applications, and Internet-based applications for video sharing and street maps, to mention a few. The phone can be extended by installing further applications, much in the same way as a conventional desktop or laptop personal computer is.

Smartphone Central Processing Unit (CPU) clock frequencies are characteristically in the range of 200–250 MHz and Random Access Memory (RAM) capacity between 64 and 128 MB. There is typically a per-process quota for memory on the order of 10–20 MB, and in contrast to conventional computers, virtual memory is not available. As power source rechargeable batteries are used with capacities between 800 and 1200 mAh. [98, 103]

Smartphone technology is evolving, and some of the figures given above are already becoming outdated, although the source is relatively recent. For instance, the more recent Dream has a 528 MHz CPU and 192 MB RAM in total, with a quota of 16 MB per process [33, 41]. However, the capacity of the standard battery is 1150 mAh, which still lies inside the range reported above.

A general trend is that the processing capabilities of smartphones are improving rapidly, as is the case with conventional computers. However, the amount of expendable energy, both in total and per unit of time, on the battery-powered smartphones is orders of magnitude smaller than on conventional computers connected to the electrical grid. While energy shortage can to some extent be offset by optimizing smartphone CPUs for processing cycles per energy unit ("MIPS per watt") , the amount of processing and other energy-consuming activities is still limited compared to conventional computers [105]. The gap is not likely to disappear, as battery technology is improving slowly [83]. Furthermore, highly energetic processing on a small device would introduce considerable heat dissipation issues.[2]

The prevalent technology for persistent storage is flash memory [85], with capacities up to 4 GB [98]. As with processing, the amount of persistent storage on smartphones has been growing rapidly. In contrast to processing and networking, flash memory requires energy only when in active use, so energy is less of a limiting factor in this case, setting smartphones less aside from desktops in this matter [97].

Besides the computing substrate, data communications are of central importance in a distributed system. On a smartphone communications is typically quite expensive in terms of energy [4, 51]. Furthermore, network usage may incur a monetary cost to the user [37]. Energy is also consumed

[1] `http://www.android.com/`

[2] In the process of writing this text, I have been making extensive use of the data connectivity of my smartphone. The energy consumption and heat dissaption issues are very apparent: the phone becomes quite hot, and it needs to be continuously connected to a charger to prevent the battery from running out prematurely.

by other components such as cameras, GPS receivers, and other sensors. However, these are not considered here, as they were not utilized in our research.

Contemporary smartphones typically include several wireless networking technologies. For services provided by a telecommunications operator, such as voice and wide-area data communications, there is an interface for second (2G) and third generation (3G) cellular networks, such as GSM [23] and UMTS [80]. This may be complemented by a WLAN interface for connecting to institutional and home networks, as well as to WLAN *hotspots*. Compared to a cellular network, WLAN typically offers higher bit rates and lower latency. On the other hand, WLAN is not as ubiquitous as cellular networks. Besides cellular and WLAN interfaces, Bluetooth is also commonly available. Bluetooth mainly provides energy-efficient short-range device-to-device connectivity at lower speeds than WLAN, and is popular for connecting peripheral devices.

The characteristics of the network connection may vary greatly depending on the network interface used and the physical circumstances [105]. Network throughput varies from some 30 kbps in the case of a General Packet Radio Service (GPRS) [13] connection up to above 1 Mbps in the case of a WLAN connection. There is also a rather large variance in latency, from some 600 ms in the case of GPRS to 10 ms when using WLAN. [98]

Furthermore, the smartphone may experience periods of disconnection, due to, e.g., radio interference, hand-off between cellular base stations, lack of coverage, or unreasonable pricing when roaming to a foreign network. *Weakly connected* is a term commonly used to describe devices connected to this kind of network, and the ability of an application to remain functional despite periods of network partitioning is known as supporting *disconnected operation*.

It is possible to network a set of devices by forming an *ad-hoc network* [86] between the devices without relying on an existing network. However, in this thesis we consider *infrastructure-based networking*, where the smartphone is connected to the Internet through a wireless *base station* (also *access point*). The base station is part of both the fixed and wireless networks, and thus acts like a gateway between these. This leads to a *two-tiered network topology*, where nodes in the fixed network share a high-bandwidth, low-latency network, while the final routing step, or final *hop*, goes over the wireless link, and is thus subject to latency, bandwidth, and connectivity issues.

Hand-off between base stations belonging to the same network access technology can usually be managed at the link-level layer in a manner that is transparent to the network layer. However, this is not the case in a *vertical handover*, where the access technology is switched. In this case the smartphone typically has to acquire a new IP address, causing existing network connections to fail. While there are proposed solutions to this problem such

as Mobile IP [47] and the Host Identity Protocol (HIP) [74], the current state of the matter is that a mobile device cannot be assumed to have a fixed IP address.

When referring to the general class of devices considered here we will use the term *mobile device*. We use this term because it emphasizes the mobility aspect, and because the energy constraint argument is to some extent also applicable to more powerful mobile devices, such as laptop computers. While we have evaluated our research on a smartphone platform, we have in general targeted this broader class, which also includes, e.g., Personal Digital Assistants (PDAs) equipped with networking facilities. Because of the need for economical use of the processing and networking facilities, the category of devices we are describing here is also referred to as *weak* or *limited* in the literature.

2.1 Designing Software for the Mobile Environment

Compared to the desktop environment, the features of the mobile environment call for a differing software design on many accounts. The limited supply of energy suggests that we minimize the amount of processing and network usage, especially since long-lasting batteries is an important selling point. However, in optimizing for energy consumption one should carefully consider the benefits of a design that weakens interoperability with existing services. For instance, in the Wireless Application Protocol (WAP) [131] efficiency was favored over interoperability with the existing Internet, leading to criticism and slow adoption [32]. We also note that because of the smaller form factor, user interface (UI) design differs considerably. However, we do not consider UI aspects here.

A smartphone is expected to be continuously powered on, rather than explicitly started up for a certain task. Thus, software may be continuously running for long durations (even months or years). This means resources need to be meticulously managed so that they do not remain allocated when no longer needed, i.e., there should not be any resource leakage. Even a small leak rate may build up to large amounts over time. In the case of memory, the problem is further accentuated by the lack of virtual memory. [103]

As there is a certain startup energy as well as energy overhead associated with using the wireless network interfaces [4, 71], it is beneficial to combine network transmissions into fewer runs of high network usage. In cases where increased computation may be used to reduce network load [7, 51], (e.g., by using data compression) energy consumption should be taken into account in addition to other factors. In particular, ongoing background communication is to be avoided [98].

Because of the high latencies of the network, an asynchronous communications paradigm is advocated to make efficient use of the network, and to avoid waiting for it unnecessarily. However, one should be aware that asynchronous programming is more demanding on the software developer than synchronous. [98]

The two-tiered network topology lends itself to the introduction of *gateways* in the fixed network that act as intermediaries between the fixed and wireless environments. The gateway approach has been proposed to improve Transmission Control Protocol (TCP) connections [11], for secure wireless messaging [51], and it is used for Web content transcoding in WAP. Security is a concern with gateways, as usually they need plaintext access to some part of the communication.

Software development for smartphones can be rather laborious compared to the desktop due to the above concerns, but also because of issues with today's development environments. Software developers are currently likely to run into discrepancies between emulated and actual environments, as well as unexpected issues and platform defects during both native and Java development [53, 98]. Therefore, it seems worth stressing the points raised in [90]: any Application Programming Interface (API) provided by the software needs to be well-designed, modular, and extensible.

Chapter 3

Data Synchronization

In this Chapter we present an overview of data synchronization. We start with more theoretical questions, such as what consistency means, and how we may reason about causality between edits made at different nodes and different points in time. We then move on to actual synchronizers, with particular attention to state-based optimistic ones, and their application to file system synchronization.

Consider a computing environment where the computational nodes $N_1, N_2, \ldots$ exchange messages over a network. Each node harbors an object ($O_1, O_2, \ldots$) and uses messages (M_{ij}) to communicate edits ($\varepsilon_1, \varepsilon_2 \ldots$) to

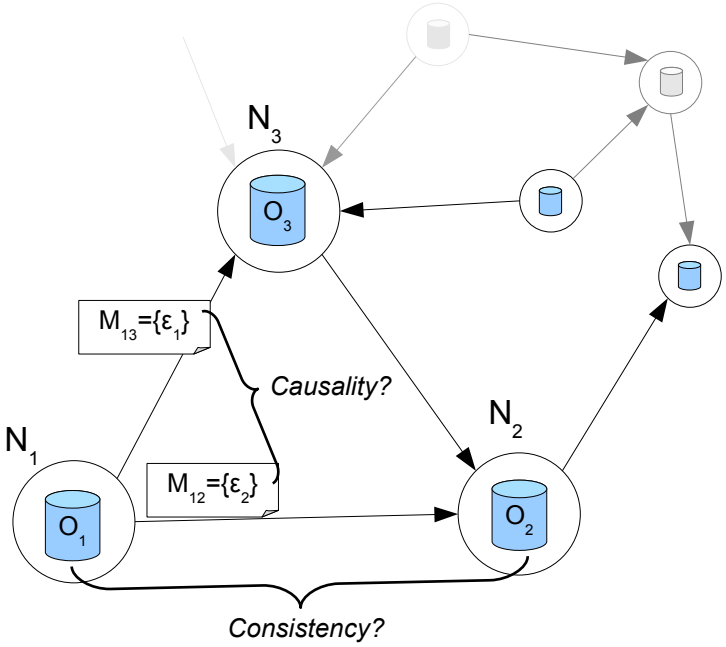

Figure 3.1: Data Synchronization

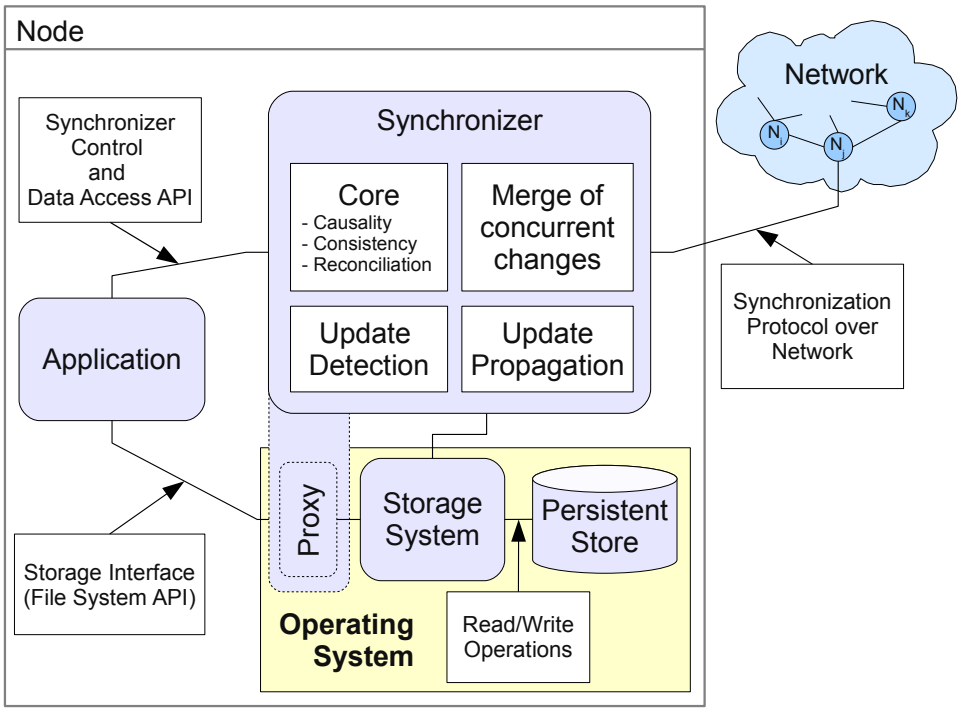

Figure 3.2: Data Synchronizer Components

objects on other nodes. The purpose of the message exchange is to synchronize the objects. As the synchronization algorithm is distributed in nature, both the actual flow of the local processes and the interaction between processes, i.e., the communications *protocol*, are of interest. This setting is illustrated in Figure 3.1, where we have hilighted a few nodes ($N_1, \ldots, N_3$) and the interactions between these.

We first describe the data objects that are subject to synchronization in Section 3.1. Then, comparing any two objects in the system, we may ask if they have the same content, i.e., if they are consistent. We can also compare edits in the system, and consider the causality between these. For instance, did ε_1 cause ε_2, and does ε_2 rely on effects of ε_1? Or, if both ε_1 and ε_2 originated from the same object, and ε_1 preceded ε_2, is ε_2 still meaningful without ε_1? Causality and consistency is considered in Section 3.2. Having considered these more theoretical concepts, we then move on to the particular type of synchronization known as optimistic in Section 3.3.

Figure 3.2 shows an archetypal data synchronizer, whose components are presented in Section 3.4 and onwards, and which we will briefly describe here to sketch an overall picture of data synchronizers in general. Starting from the data, we have one or more data objects in some persistent store which accepts read and write operations from a storage system such as a "files-and-directories" hierarchical file system. Applications use

the stored objects, and access them using a model that is supported by the synchronizer. In some cases this means to use the storage interface directly, and in other cases it means using a data access interface provided by the synchronizer. Some synchronizers also act as a proxy for the storage interface in order to monitor activity and alter the behavior of the original interface. The optional nature of this functionality is illustrated by a dashed boundary for the proxy component in the Figure. The synchronizer may furthermore provide an interface that applications can use to control the synchronization process.

The interface used by the synchronizer to communicate changes with other nodes consists of the synchronization protocol and its serialization over a network transport. Changes are readied for transmission using the update propagation mechanism, which for instance compresses the changes to reduce network use. To obtain what changes to an object there were, the synchronizer uses an update detection mechanism. For instance, changes may be detected by comparing past and present revisions of an object.

At the core of synchronization we have the causality (versioning) and consistency model, and reconciliation of local and remote edits. The model typically captures the causality between edits, and determines how communication is scheduled and how data objects may be accessed in order to remain compliant with the consistency guarantees that the model provides. Reconciliation is the logic that applies local and remote edits in a consistent manner across nodes with support from the causality and consistency model. We consider reconciliating edits that happen concurrently explicitly in the merging subsystem.

Note that we here for the sake of clarity consider only a single object per node. The generalization to more objects is straightforward.

3.1 Data Objects and Synchronization

We start by establishing some terminology and by discussing properties of the data that we synchronize. We use the term *object* for the pieces of data that are subject to synchronization, and allow the data contained in an object to change over time. We use the term *state* for a snapshot of the content. What constitutes the objects of a synchronizer varies from implementation to implementation, so it is hard to give a general and yet precise definition. One possible definition, which we have used here, is that the object is the smallest unit of data to which the synchronization process can be applied. For instance, in a file synchronizer, the same overall steps are used for each file, and thus the file is the object. Another example is the SyncML protocol [117] which synchronizes named collections of data records (e.g., calendar entries). Here the collections become the objects.

A synchronizer adds propagation of changes between objects. A group of objects that exchange updates with each other through the synchronization mechanism are known as *replicas*. Sometimes it is useful to distinguish the objects as they appear outside the synchronizer compared to how they are viewed from within. We use the term *user object* for the former case, where the users may be both humans (also known as *end users*) as well as other computer applications. For instance, in the case of file synchronization, the user objects are files.

To be able to access a particular object among others, we need to attach a discriminative *name* to the object. Persistent and unique names which may be used to identify resources and communication endpoints are crucial for the organization and access of resources. This is particularly true when managing data in a distributed system [66].

We can distinguish a few common ways of naming user objects. One way is to attach a distinct name to each replica, and another one is to attach a name to each group of replicas. In the latter case the distributed nature of the object is hidden, as one is no longer able to name the individual members of the group. Instead, the synchronization system becomes responsible for channeling access to an appropriate replica, and for masking situations when access becomes inhibited. In the former case, we lose the convenience of being able to address the replicas using a single group name. The individual replicas become visible, for better and for worse.

Object names may encode information on object type, access protocol, and location in a distributed system. A common case in this category is to use Uniform Resource Identifiers (URIs) [9] for object names. Another option is to name objects by their state. To keep names manageable in practice in this case we use a *fingerprint* [91] of the state, typically in the form of a secure hash. The advantage of this scheme is that the name of any object can be computed locally without coordination. URIs have been used in [116], and state-based names were used in [96, 19, 109].

Both synchronizers and users may utilize or require some particular *structure* for the content of each object. In [70], four different levels of structural awareness are presented: textual, syntactic, semantic, and structural. Consider an object containing program source code. A *textual* synchronizer processes it a sequence of characters without any particular meaning. On the *syntactic* level, the syntax of the programming language may be utilized. Finally, on the *semantic* and *structural* levels, we may use knowledge of, e.g., the intended behavior of the program to prevent synchronization of changes that would introduce erroneous behavior.

We note that it can be detrimental if a synchronizer requires a particular structure for objects that is not naturally supported by the user. Consider a system that requires content to be XML data. In this case, synchronizing a binary image in, e.g., JPEG format [46], is not immediately supported,

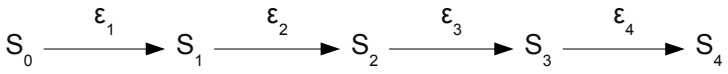

Figure 3.3: Editing the state of an object

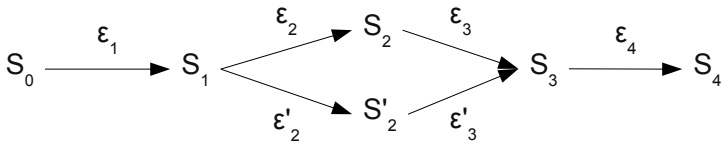

Figure 3.4: Concurrent editing of an object

unless one goes through the trouble of devising an XML encoding for JPEG images, and adds support to this in applications.

The term *metadata* is commonly used for data describing other data, with the metadata typically requiring significantly less space compared to the data it describes. Examples include dates relevant to the creation and modification of the data, format of the data, size in bytes, and keywords. Some synchronizers are designed for synchronizing metadata only [116].

3.2 Causality and Consistency

A process where an entity changes the state of a data object through a sequence of edits is depicted in Figure 3.3. Starting from an initial state S_0 we apply the edit operation ε_1, which results in state S_1. State S_1 then yields S_2 through the edit ε_2, and so forth. The graph of states and transitions through edits is the *history* of the object. A state with no successor state (S_4 in the Figure) is known as a *current* state of the object.

We may extend the example so that several edits are applied to the same state. In this case, a state may have several successor states, corresponding to alternate *branches* in the object's history. However, we typically aim for *consistency*, where the object has only one current state. This is achieved by *reconciling* two or more states into a common state with the help of reconciling edits. The reconciling edits typically incorporate the changes from the other branches.

Branching and reconciliation is illustrated in Figure 3.4. Here, S_1 is edited in two different manners with ε_2 and ε'_2, yielding two current states S_2 and S'_2. These are then reconciled into the state S_3 with the reconciling edits ε_3 and ε'_3 , where ε_3 incorporates ε'_2 and ε'_3 incorporates ε_2. Typically S_2 and S'_2 would reside on different nodes, and the node with S_2 would execute ε_3 to reconcile, while the node with S'_2 would execute ε'_3.

Multiple current states of an object arise when an object is edited in different manners. It is then the task of the data synchronizer to restore consistency by exchanging states and edits between nodes. The synchronizer thus needs to be able to reason about these, and answer questions like: How do we order edits and states received from remote nodes? What were the edits to a state? Which states may we apply a remote edit to? How do we know when a replica is consistent?

3.2.1 Causality

To help us reason about these matters we introduce the *happens-before* relation, as defined by Lamport in [58]. Consider a distributed system consisting of an arbitrary number of concurrent processes. Within each process, an ordered sequence of *events* occurs, such as the transmission and reception of messages. Given two events a and b, Lamport defines that *a happens-before b* if

1. a and b are events within the same process, and a precedes b in the process' sequence of events, or
2. if a is the sending of a message, and b is the reception of the same message. In this case, a and b may occur in different processes.

Furthermore, if a happens-before b and b happens-before c, then a happens-before c, i.e., the relation is transitive. We denote that a happens-before b with $a \rightarrow b$, as is done in [58].

We stress that despite its name, the happens-before relation has very little to do with the time at which the event occurred. Rather, a happens-before b expresses a relationship of *causality* between a and b. More exactly, a happens-before b means that it is possible for a to have causally affected b.

Consider the events in the processes A and B depicted in Figure 3.5, where message exchanges are drawn using arrows. Here, a_1 happens-before both a_2 and a_3, by the definition of happens-before inside a process. The events a_1 and b_1 correspond to the transmission and reception of message μ_1, and hence $a_1 \rightarrow b_1$. Since $b_1 \rightarrow b_2$, and $b_2 \rightarrow b_3$, we may deduce that $a_1 \rightarrow b_3$ using the transitivity of the relation.

In the Figure, neither may a_2 have causally affected b_2, nor could b_2 have causally affected a_2, since no information of either event had arrived at the opposite node. In such a case when there is no possibility of causal effect of one event on another, we say that the events are concurrent. Using the happens-before relation, if both $a \not\rightarrow b$ and $b \not\rightarrow a$, the events are *concurrent*.

In data synchronization we particularly consider events that are edits on objects. A reasonable assumption to make is that each new edit on an object is affected by a subsequence of previously applied edits to that same

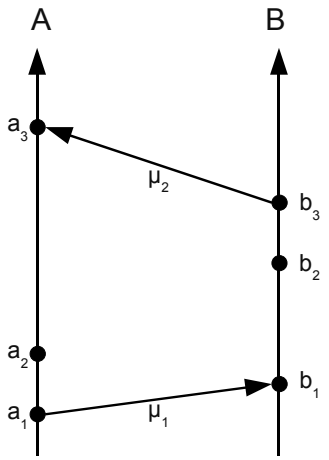

Figure 3.5: Events is a distributed system

object. For instance, when editing a text document, changes are typically made based on existing content in the document. Note that in some cases the sequence of edits causally affecting a new edit may be empty, as for instance in the case of recording the outcome of tosses of a fair coin. Furthermore, there may be information external to the system that affects a new edit, such as an author's knowledge and intentions.

Let us consider the creation of a new edit ε. Setting aside any communication not modeled by the system, the *maximal* set of edits that may causally affect the edit ε is exactly the set of edits that happened-before ε.

We now see why the happens-before relationship plays an important role in data synchronization. If we receive an edit ε for an object o, and all previous edits on o happened-before ε, we know that ε positions itself last in the sequence of edits on o. If there is an edit ε' on o that is concurrent with the creation of ε, ε may no longer be valid, as it cannot possibly have taken ε' into account. In such cases, we need to reconcile the effects of ε and ε'. We may also find that we have not yet received all edits that happened-before ε, in which case it is not necessarily correct to apply the edit, as the current sequence of edits may have caused another edit than ε (e.g., one that is in transit but has not yet been received).

Of the set of edits that happened-before a particular edit, a synchronizer typically does not know the exact subset that affects the edit. However, we may make the safe assumption that an edit is affected by any edit that happened-before it. Encoding the happens-before relationship between edits, or an approximation thereof, therefore takes a prominent place in the design of a data synchronizer.

The *Lamport clock* [58] is a scalar-valued function $C(\cdot)$ on events so that if $a \rightarrow b$ then $C(a) < C(b)$. Such a clock can be implemented using an integer counter, which is incremented at the nodes of the system in accordance

with the so-called clock condition. Due to the compact representation of the Lamport clock, we could conceivably stamp the clock value onto edits at creation time, and then use that value to compute the happens-before relationship between two edits. For instance, if $C(a) = C(b)$, it follows that a and b are concurrent. We could also approximate that $C(a) < C(b) \Rightarrow a \rightarrow b$. However, since this is not necessarily true, in such a system we would risk falsely deducing that the effects of a were already taken into account into b. For instance, if we in Figure 3.5 assign clock values according to the position of the event on the vertical axis, the clock condition is satisfied, and $C(a_2) < C(b_2)$. However, as pointed out previously, $a_2 \not\rightarrow b_2$.

A popular method for determining the outcome of synchronization is to use the state with the most recent change time [102]. Extending the method with a deterministic ordering for equal timestamps yields the so-called Thomas' write rule [123]. However, in many applications, the change time has no correlation with the happens-before relation. In such cases, the change time provides no help in determining the causality between edits, and using it for this may lead to loss of data. As an example, consider two authors who download the same text onto their laptops, and then continue working on their copies in isolation. When the authors meet again, it makes little sense that they would consider the copy that has the most recent timestamp as the "latest" revision. Note that the situation does not improve even if the system clocks in the laptops were synchronized to some arbitrarily small tolerance.

A *lost update* occurs when an edit is erroneously discarded or otherwise fails to propagate correctly. As this is a form of loss of data, it is generally considered undesirable to allow lost updates in a data synchronizer [107, 97, 95, 36]. For instance, in the example above, a lost update would occur due to the failure of the timestamp-based scheme to detect concurrent edits.

The *version vector* [84] (also *vector clock*) is a compact data structure for encoding the happens-before relationship. The components of the version vector $\mathbf{v} = (v_1, \ldots, v_n)$ are monotonically increasing numbers, where the node n_i is associated with the component v_i. We assign version vectors so that the next edit on n_i gets the vector $\mathbf{v}' = (v_1, \ldots, v_i + 1, \ldots, v_n)$, where $\mathbf{v}$ is the version vector of the preceding edit. That is, the components v_i act as local version counters for the nodes. The version vectors $\mathbf{v}^a$ and $\mathbf{v}^b$ assigned to the edits a and b now allow us to determine if either edit happened-before the other according to $a \rightarrow b \Leftrightarrow \forall i : v_i^a \leq v_i^b$ and $\mathbf{v}^a \neq \mathbf{v}^b$.

Version vectors have been used in several systems [108, 87, 88, 37]. The use of one vector component per node is problematic in some applications, especially in cases where the set of active nodes changes frequently. Subsequent work [127, 3] has addressed this to some extent.

The *hash history* [48] is a structure for encoding happens-before that scales with the number of states, rather than the number of nodes. Here,

the history of previous states is communicated along with the current object state in the form of a graph of state fingerprints.

3.2.2 Consistency

Making the replicas of an object consistent, either *de facto* or in appearance, is challenging. This is especially the case if message passing between nodes is constrained in some ways, which is often the case. Therefore, some data synchronizers relax the requirement for strict consistency between replicas. The tradeoff for this increased relaxation is some exposition of the differences between the replicas of an object.

At the strictest level of consistency there is single-copy consistency, or *linearizability* [39], where each operation on each node observes values and has effects that are equivalent to a serial run of the operations on a single object. Implementing single-copy consistency in a fully decentralized manner typically amounts to establishing distributed consensus in an error-prone environment, which requires relatively complex protocols (e.g., [96]), and is problematic even in theory [26].

A popular alternative is to offer *eventual consistency*, where the replicas will converge on the same state after some amount of additional messaging after update activity has ceased [102, 122]. Eventual consistency is, however, not always a strong enough guarantee for applications to operate correctly, or to isolate users from anomalies due to the distributed nature of the system. For instance, the state of an object may appear to revert to a previous value, if suddenly a less recent replica is taken into use.

The Bayou [87, 122] system provides several levels of consistency to application sessions, so that the application may use the most appropriate one. The levels model different tradeoffs between consistency and availability. For instance, the "Read your Writes" level guarantees that reads in a session see any previous writes within that session, whereas the "Monotonic Writes" level guarantees that writes from a session are seen in order on all replicas. The latter condition constrains the set of replicas capable of serving the request more than the former, decreasing the availability of the object. The OceanStore [96] large-scale storage service provides both sessions with single-copy consistency and sessions with weaker guarantees.

Content addressable storage systems identify an object by its content [109, 96], as expressed in condensed form with a fingerprint. With this method the content of the object is consistent by definition. However, maintaining consistency does not become fundamentally easier in such systems, rather it is trivially solved by making each state an object of its own. The changing association between the states that form the history of an object still needs to be made consistent.

Consistency may also be required across objects. A pertinent example is a text file and a file indexing the words of that file. In this case, only some

combinations of object states constitute a consistent state for the indexed text. Consistency across objects typically requires some notion of a transaction [64, 112]. We have not considered consistency management across objects in this thesis.

Data synchronization research until the late 1980s focused mostly on techniques for single-copy consistency, where access to a replica was prevented unless the replica was provably up-to-date. In particular, systems where one replica was designated as the *primary replica* were used. The primary replica controlled access to the object, thus acting as a single point where accesses were linearized. In case of failure there was a protocol for electing a new primary replica. [107, 102]

Guaranteeing consistency in a design with a primary replica becomes less challenging, as does maintaining the happens-before relationship: a simple counter at the primary replica suffices. However, channeling all accesses through the primary copy does not scale well with increased load. Furthermore, when clients use locks to obtain exclusive access, failing to release a lock may lead to the entire system becoming unavailable.

The assumption that any replica not provably consistent is unsuited to serve object accesses gives rise to the term *pessimistic* for describing these techniques. Another name is *traditional* techniques, to set them aside from more modern approaches, such as the optimistic approach. We will discuss this approach next.

3.3 Optimistic Data Synchronization

The *optimistic* model for data synchronization overcomes many limitations of the traditional synchronization model by trading off strict consistency guarantees for increased availability and flexibility [102, 107]. The adjective "optimistic" refers to the assumption that accessing data without strict consistency guarantees will rarely, if ever, be problematic. In optimistic systems we typically allow replicas to be temporarily updated in a diverging manner.

For data synchronizers built on the optimistic principle, a set of common aspects have been identified in [6, 102], which we will present here briefly. Some of the aspects are also pertinent to traditional systems. However, here we focus on their manifestation in optimistic systems.

Update Detection In many practical applications edits are not readily available for propagation by the synchronizer. Rather, a mechanism is needed to establish *if* a replica has been edited, and sometimes also *how* it was edited.

Update Propagation Update propagation is the communication of object edits over the network. It needs to be designed with care, so that en-

ergy is not needlessly consumed and the data synchronizer remains agile despite weak connectivity. Update propagation between nodes may be restricted by some communications topology so that a given node may not be able to communicate directly with any other node in the network.

Reconciliation To counteract replica divergence we need a mechanism for reconciling concurrent edits. This includes detecting concurrency, e.g., using the mechanisms for capturing happens-before mentioned previously. The reconciliation mechanism may classify some concurrent edits as irreconcilable, in which case they form a conflict.

State or Edit A system typically stores either object states S_n or the edits ε_n between states, based on which we can classify a system as either *state-based* or *edit-based*. It should however be noted that state-based systems may compute edits for reconciliation and update propagation purposes, and that edit-based systems may maintain a state for improved performance.

Stream or Snapshot In general, data synchronization may be implemented as a continuous process, or as periodic runs of limited duration. We refer to these as the *stream* and *snapshot* approaches respectively (also *immediate* and *periodic* [37]). Typically, streaming synchronizers are edit-based, while snapshot-based use state. Note that our definition of streaming synchronization does not exclude disconnected operation. Disconnection merely means that the process temporarily halts.

Readers and Writers Among the replicas of an object, only some may be directly written to, while the rest receive updates from the writable replicas only. In such cases, the writable replicas are known as *master replicas*. In some cases the writable objects are co-located at a set of *master nodes*.

Commitment A common requirement is to have identifiers that name a state across all replicas[1]. We refer to the process of establishing these identifiers as *commitment*. For instance, we may identify a state with a revision number, and expect that revision number to identify the same state regardless of which replica we use.

The need for commitment arises from both user and system aspects. Users typically want to be able to name a specific state, as in "the revision from yesterday". On a system level, commitment helps manage the amount of edits and states that need to be retained. A straightforward method to implement commitment is to have a primary replica that assigns identifiers to

[1] As defined in [102], commitment does not include identifiers for the established states. However, we suggest that these are a relevant part of the concept.

states [87]. Uncommitted updates are commonly referred to as *tentative* and a system may allow access to tentative updates in additional to committed ones [87, 96].

We discuss update detection and propagation and reconciliation in more detail in the following.

3.4 Update Detection and Propagation

The first step in synchronization is to perform *update detection* where we identify the changes to propagate. This step may be trivial in some cases, e.g., if the synchronizer is able to monitor data change operations, whereas some application domains require considerable engineering effort.

Update detection typically need not result in a minimal set of changes. Rather, we may tolerate false positives in the form of changes that have already been synchronized. However, propagating such changes consumes networking and other resources needlessly, and may cause false detection of concurrent updates that complicate reconciliation. Thus, in general false positives are to be avoided.

Update detection is particularly significant in synchronization of file systems, as these typically only maintain state by default [92]. Accurate tracking of file system activity is possible by implementing a synchronizer that also acts as the provider of the file system. Application file operations are then intercepted by the synchronizer, and may be logged. The synchronizer may in turn use another file system for storage, in which case we say that the approach is that of a *layered* file system. The file system provider approach relies on functionality beyond the traditional file system interface, and may require system-level functionality and a continuously running process to provide the file system. Examples of approaches providing a file system include Coda [108], OceanStore, and Ficus [88].

Another type of file synchronizers is written as ordinary applications that use the standard file system interface. In this case the file system needs to be traversed in order to find modified objects. Typically, file metadata, such as time of last modification, length, or inode number[1], is used to quickly determine if a file has changed, although such an approach may miss some uncommon cases [6]. A safer approach is to examine the full contents of the file, but this may be computationally expensive. Synchronizers in this category include the Concurrent Versions System (CVS)[2], Rsync [128], and Unison[3]. Both Rsync and Unison allow the user to specify different update detection algorithms to trade off accuracy for speed.

[1] On Unix-like systems, this number identifies the storage object that the file name is bound to.

[2] `http://www.nongnu.org/cvs/`

[3] `http://www.cis.upenn.edu/~bcpierce/unison/`

Change detection may in addition to finding out *which* objects were modified also establish *how* the states of the objects were changed. For instance, in addition to knowing that a text file has been changed, we may also detect that the change consisted of, e.g., removing the last line from the file. Detecting changes with a finer granularity than whole-object may save bandwidth during updated propagation and can also improve reconciliation.

In addition to monitoring change activity, change information may also be obtained by comparing the current object state S to a previous state S' maintained by the synchronizer. This process is known as *differencing*, or *diffing*, as it is more commonly called. The output of the diffing process is known as a *delta* (also *diff* and *patch*). The delta may then be used to reconstruct S using S' in a process known as *patching*.

A well-known pair of tools for diffing and patching text files is the Unix `diff` and `patch` tools which are based on the text differencing algorithm in [77]. For binary data, there is for instance the `xdelta` tool [67]. CVS computes changes to text files using the `diff` algorithm, and Rsync is able to compute changes on arbitrary binary files. In the Rsync algorithm, S is constructed using references to subsequences of bytes from S' along with subsequences of S not found in S'. This same general idea is used in `xdelta`. There are also diffing and patching algorithms specifically tailored for the XML format, which we will return to in Chapter 4. The use of deltas in the context of Coda has been studied in [38].

The detected changes may sometimes be optimized by leaving out those that only contribute to short-lived object states that need not be synchronized. For instance, temporary files are commonly created, and these files typically need not be synchronized. Such files may be optimized away if we buffer changes locally before propagating them, and then prune any changes that cancel out or are obsolete from the buffer before propagation. The buffer is typically constructed so that a change must age a certain amount before exiting. When the delay time of the buffer exceeds the life span of a transient state, that state may be optimized away. This technique is used in [108].

3.4.1 Update Propagation and the Synchronization Protocol

With update propagation we understand the synchronizer's mechanism for transmission and reception of updates over a network transport. We may classify the update propagation mechanism according to the *synchronization topology* (also *reconciliation topology* [37]), which describes which nodes may exchange updates directly, whether one-way communication is sufficient, and how the messaging channel is utilized over time. We are also interested in how updates may be transformed and packaged to make optimal use of channels with a limited bandwidth and high latency.

The *synchronization protocol* defines how the synchronizer exchanges messages with other synchronizers. The main, and sometimes only, task of the synchronization protocol is to implement update propagation. However, a synchronization protocol may also include such functionality as object locking [108], information exchange for data compression [128], distributed agreement [96], or even data search operations [117]. Some synchronization protocols take a very generic view of the data they synchronize, whereas others are more specifically tied to particular data structures.

We may observe some differences in the message exchange patterns exhibited in synchronization protocols. Messages may be exchanged in a synchronous or asynchronous manner. In the *synchronous* pattern messages are exchanged as a dialogue of requests and replies, where the transmitting party explicitly waits for the response before proceeding. In the *asynchronous* case, on the other hand, processing proceeds in response to incoming messages, along with other activities. If the protocol is *one-way*, no reply (synchronous or asynchronous) is required to transmitted messages. A special case of one-way messaging is to let updates be propagated by means of storing and retrieving them from some storage media. Another aspect is whether message traffic is typically *bursty* with stretches of idleness interrupted by periods of high network utilization, or if it is more of a *continuous* nature.

Bayou features an asynchronous model which supports continuous propagation of updates. On the other end of the spectrum, there is CVS, which relies on synchronous bursty communication. Coda lies in between, with both synchronous and asynchronous message exchange used during synchronization. Depending on which state Coda is in, network traffic may be bursty or continuous and frugal in use of bandwidth [108].

In a fully unconstrained synchronization topology each node may exchange updates with every other node. This is known as *epidemic propagation* [22]. However, a fully unconstrained topology is not always desired, as we may improve change propagation time and reduce complexity by introducing a more regular topology [102]. For instance, in the *star* topology all updates pass through the central node, which introduces a fixed-length path for updates, and simplifies synchronizer design. This is because the central node can easily enforce linearizability, perform reconciliation in a non-distributed manner, assign state identifiers, and maintain a natural "most recent" state of objects.

To overcome the reliance on a single central node, we may replace it with a central network of nodes, yielding a *two-tiered* topology. Typically, the inner tier will act as a single logical entity to the nodes in the outer tier. As an example of a two-tiered topology, there are high-traffic Web sites, where the inner tier consists of a group of servers, and OceanStore, where the inner tier consists of a group of nodes that establish committed states of the object.

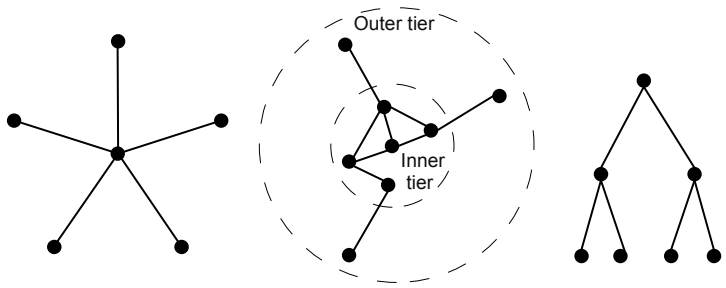

Figure 3.6: Star, two-tiered, and tree synchronization topologies

In the *tree topology*, each node that is not a leaf acts as a central node for its children, thus providing scalability by successively reducing complexity on each level of the tree. The Domain Name System (DNS) [72, 73] is structured using a tree topology.

Figure 3.6 illustrates the star, two-tiered, and tree topologies. Note that the topology of the actual messaging network may in practice further constrain the pair of nodes that can communicate compared to the idealized view of the synchronization topology.

As mentioned when we considered update detection, one step towards reducing network bandwidth requirements is to use a change detector which yields compact changes. The next step to further reduce the size of changes is then to use algorithms for data compression and efficient encoding. Furthermore, if connectivity is weak, latency has a significant impact. In this case we may *batch* (also *bundle*) several changes together, so as to eliminate unnecessary network roundtrips.

Rsync, Unison (which uses the Rsync algorithm), and CVS all include functionality for compressing network communications. [76] presents a method based on maintaining a cache of file fragments, and then using these fragments to construct the object state. [59] explores the idea of transmitting shell commands that generate the object state, rather than the state itself. Using batching to reduce the impact of network latency was one of the improvements introduced in the 1.1 version of Hypertext Transfer Protocol (HTTP) [25], as well as in version 4 of the Network File System (NFS) [113].

3.5 Reconciliation

In the *reconciliation* phase of synchronization we integrate into a replica changes from one or more other replicas. Alternative terms used in the literature are *integration* and, also, merging. We emphasize that here the term includes both the case when changes are classified as concurrent and the case when changes may be ordered. In particular, this includes the case

where a received change is simply appended to the local history of edits. A *conflict* arises when the reconciliation mechanism, unaided by human intervention, is unsuccessful in integrating a remote edit into the existing history, thus potentially leading to a lost update.

We will discuss reconciliation from the perspective of a process executing on a given object. The process receives an edit ε_m from a remote replica, which it reconciles with the current state and the edit history of the object. State-based systems are modeled by considering edits ε_m for which $S_i \xrightarrow{\varepsilon_m} S_m$ for any state S_i. Although we look at one operation only, in practice one will usually receive several edits per communication. However, this typically reduces to reconciling the operations one by one, or by considering ε_m to be a cumulative edit capturing the effects of all edits involved.

When determining at which point in the edit history we want to insert ε_m we typically consider the happens-before relationship between ε_m and the part of the edit history that is locally known. If we find that the happens-before relation yields an unambiguous position for ε_m among the existing edits, and the edits before that position satisfies the dependencies of ε_m, we may insert ε_m at that position. In state-based systems, this translates to either setting the state to that implied by ε_m (if the previous state happened-before ε_m), or ignoring ε_m (as the edit that ε_m happened-before translates *any* state to the current state).

In contrast to considering ε_m dependent on all edits that happened-before it, we may also encode the *dependencies* (also *constraints*, *preconditions*) of the edit explicitly, and then consider the edit applicable at positions in the local history where the dependencies are satisfied. This approach, which was used in [55, 87, 96], has the advantage that it may be easier to find a valid position for the edit in the edit history. This is because stating explicit dependencies allows us to set aside any unnecessary dependencies implied by an overall generic dependency model. On the other hand, annotating edits with an explicit dependency set can be quite burdensome for the application, as well as hard to do correctly [102].

The context diff format used with the `diff` and `patch` tools provides an example where explicit dependencies for edits are given. Here, changes are expressed so that context from the original unchanged text is included, in this case in the form of original lines of text around the change. A change is applicable to an object state if the context of the change is present in the state, but otherwise no conditions on change order etc. is imposed.

3.5.1 The Two-step Reconciliation Process

A rather common approach to reconciliation is to use what we call the *two-step reconciliation* process. In the first step the position of ε_m with respect to the other edits is either determined, or ε_m is classified as concurrent. In

the latter case, the second step is executed, where the concurrent changes are reconciled in a *merge* process. Ordering in the first step is typically attempted using the happens-before relationship, or the part of it which is known.

To perform successful reconciliation in the second step the merge may process the object in a more fine-grained manner, use different and more sophisticated methods than the first step, and utilize knowledge of the object structure. Note that while we here use the term "merge" to denote the second step in two-step reconciliation, "reconcile" and "merge" are often used interchangeably and sometimes with narrower scope, so that reconciliation entails only the integration of concurrent changes.

Two and three-way merging and the use of diffs and patches are well-known state-based merging strategies. On the edit-based side a commonly used method is that based on the Operational Transformation (OT) approach, where edits are transformed to take into account the effects of other concurrent edits [115]. We consider state-based strategies here, as that has been the focus of our research.

Let S_x and S_y be the states to merge. In basic *two-way merging* [70], the states S_x and S_y are compared for differing regions, and for each such region, the alternative from either S_x or S_y is used. Basic two-way merge has a limitation, however, as it cannot tell which of the alternatives should be used without some auxiliary information. For instance, if the differences consist of alternative records, and timestamps are available as auxiliary information, the timestamps could be used to choose the more recent alternative.

A particularly interesting piece of auxiliary information is the most recent common ancestor (MRCA) state in the history of S_x and S_y, which we denote S_b. By comparing S_x and S_y to S_b we can deduce which change alternative represents an incremental change from S_b. This is the basic idea behind *three-way merging* [70]. Comparing the MRCA to other alternatives for auxiliary information, it seems advantageous that it is a state, just as S_x and S_y, and therefore does not introduce any additional data structures in the merging process. In three-way merging we use the term *base state* for S_b and the states S_x and S_y are known as the *derived states*.

In *diff/patch merging* we either apply (patch) the diffs between S_b and S_x to S_y, or vice versa. The method is commonly used in an ad-hoc fashion during software development, and positions itself between two and three-way merging in terms of capabilities.

Two-way merging for text is implemented in such tools as the `ediff` mode in the Emacs editor[1] and in the document compare functionality in

[1] `http://www.gnu.org/software/emacs/`

word processing software such as Microsoft Office[1] and OpenOffice[2]. The Unix `diff3` tool[3] is a widely used three-way merger for text files. Darcs [99] is a distributed revision control system with diff/patch-based merging that includes a special "token replacement" patch. We consider XML merging in Chapter 4.

3.5.2 Object Life Cycle Edits

The set of edits may include operations that cause objects to be created and deleted. When an object is deleted, the goal is to reclaim all storage space required by the object, thus necessitating eradication of all information pertaining to the object, including the fact that the object has existed. This is for instance how most file systems work, as keeping even the smallest amount of information after object deletion would inevitably lead to storage space exhaustion after a sufficient number of create and delete cycles.

Interestingly, it turns out that implementing complete eradication for optimistically synchronized objects is harder than it may initially appear. To see why this is, consider the following scenario, where the edit ε_0 signifies object creation, and the edit ε_∞ complete eradication of the object. Suppose we delete the object locally by issuing ε_∞. The complete eradication criterion now requires that we delete all knowledge of the object, i.e., its content, edit history, any metadata, and the fact that we knew it existed. Now, due to the optimistic nature of the system, it is possible that we receive a change ε_n from another node pertaining to the same object. How should this change be processed?

Suppose ε_n happened-before ε_∞, which means that the object should stay deleted. However, having lost the edit history of the object we cannot establish this. This could potentially be solved by defining that any other operation happened-before ε_∞. This, too, leads to issues, as it is in fact possible that ε_n was concurrent with ε_∞, and ε_n would become a lost update. Also, how can we create an object if ε_∞ is implied by the absence of the object? Assuming the opposite, that ε_∞ happens-before any other operation, unfortunately means that receiving ε_0 would reinstate the object, in which case ε_∞ becomes a lost update.

The problem occurs in even simple synchronization models, such as pairwise file synchronization where any file not on either node is propagated to the other node: deletion on one node becomes indistinguishable from insertion on the other node, as there is no timestamp for the deleted object to inspect. An early observation of this *create/delete ambiguity* is due to [27].

[1] `http://www.microsoft.com`

[2] `http://www.openoffice.org`

[3] `http://www.gnu.org/software/diffutils/`

An alternative to deletion as complete eradication is to use *tombstones* [36] (also, *delete markers*). A tombstone discards the object state, but records the fact that the object has been deleted and information to resolve the happens-before relationship between the deletion and subsequently received edits. This allows object deletion to be managed in a manner similar to other object edits. The downside is that tombstone maintenance will require a small, but yet monotonically increasing amount of storage, unless tombstones are removed using some mechanism. One approach is to automatically discard tombstones after some specified time longer than the maximal estimated update propagation time [102], but this is not guaranteed to be correct. For ensured correctness, distributed garbage collection algorithms may be used, such as those described in [36]. Tombstones were used in, e.g., [88, 94], and more recently in [82, 97].

3.5.3 Some Observations on File System Reconciliation

File system synchronizers implement reconciliation of file system edits as an integral part of the synchronizer. However, the reconciler typically excludes merging of concurrent edits to the same file, and either classifies these as a conflict, or delegates them to a merger external to the synchronizer. External mergers are for instance used in Rumor [37] and Coda.

As the file system interface is unsuited for capturing operations native to the data structure stored in the file (as compared to operations on the byte sequence encoding of the data structure), the external mergers are typically state-based, while the overall reconciliation mechanism may use file system change operations, too. For instance, Coda natively implements a reconciler for operations such as directory entry create, delete, and update, while its application-specific mergers operate on file states. An example of a fully state-based approach to file-system reconciliation is [6].

Some file synchronizers have facilities for reconciling moves (renames), i.e., the file system operation where the hierarchical name of a file system object changes. Recognizing file moves helps reduce bandwidth required by update propagation, as the same object content can then be migrated locally from one path to another, rather than having to be transmitted over the network. In situations where the end user's input is needed for conflict resolution it is also beneficial to be able to present changes using moves rather than insert/delete pairs [92].

Chapter 4

Synchronizing XML

Having considered synchronization in general, we will now move on to synchronizing XML data. The Extensible Markup Language (XML) [143] has in the past decade become a *lingua franca* for information exchange. The comparatively large amount of structure exposed when using XML, as well as its widespread deployment motivates us to consider the problem of synchronizing data encoded as XML. Since the structure of XML is a tree, we find that the problem on a more general level is that of synchronizing tree structures.

Synchronizing XML data as opposed to an opaque sequence of bytes allows us to utilize the structure of the data exposed by the XML encoding. In particular, when merging states, we may process the object at a finer granularity than an all-or-nothing approach. As there exist well-established ways of querying and transforming XML [135, 144, 136], we may also consider synchronizing only a subset or a transformed variant of the object. Furthermore, during update propagation we may use XML-level information to, e.g., reduce resource usage [57], or to eliminate some changes that do not affect the content of the object.

We consider merge strategies for XML which are applicable to object states. Update detection is thus called for to deduce edits between states, and to be useful to the merger, the edits need to be meaningful at the XML level. Hence, XML differencing arises as an important topic in state-based XML synchronization. Differencing is also a vehicle for implementing recall of past revisions, which is of use in synchronization, too [68]. Note that the choice of data format (XML in this case) will in general not affect the reasoning about causality between changes. Hence, XML-awareness is ideally only required when merging concurrent edits, whereas other parts of the reconciliation method (e.g., ordering changes, detecting concurrency) remain unaffected.

While we do not consider synchronization where objects are related by transformations or queries further in this thesis, we note that support-

```
<p>
  This thesis was written using the open-source
  <i>
    document processor
  </i>
  <a href="http://www.lyx.org">
    LyX.
  </a>
</p>
```

Figure 4.1: A simple XML document

ing such functionality is often mainly a matter of introducing a suitable mapping between the full and transformed content at an appropriate point in the synchronization process. However, constructing such a mapping, known as the *view update problem* in the bi-directional case, is a large topic of its own. The problem in the unordered tree case is investigated in [29].

In the following, we start by introducing XML. This is followed by XML differencing and merging methods. XML differencing and merging are considered as we have identified these as the parts of synchronization where XML-aware processing matters.

4.1 The Extensible Markup Language (XML)

Along with the rise of the Word Wide Web and the HyperText Markup Language [133] (HTML) in the mid to late 1990s there was a need for a markup language with general applicability and which would be suited for the Web. The existing Standard Generalized Markup Language (SGML) [45], of which HTML is an application, was deemed too complex, and thus unsuited. However, SGML was still considered a good starting point, from which unnecessary complexity could be removed. The result was the Extensible Markup Language, XML. [34, 143]

A simple XML document is shown in Figure 4.1. As is the case with HTML, XML is a text-based format, which means it is relatively easy for humans to author and read. Document structure is introduced by using matching pairs of opening and closing *tags*, where in the basic case the opening tag is of the form <*element-name*> and the closing of the form </*element-name*>. Together the matching tags form an *element*, whose name is *element-name*, and whose enclosed content (other elements or text) is the part of the document between the opening and closing tags. The opening tag may also include a set of *attributes* with associated values. The outermost element of an XML document, of which there must be exactly one, is known as the *root*.

Thus, in the Figure the and tags delimit the p element, whose content is the text "This...", the element i, and the element a. The a element has an attribute href, whose value is http://www.lyx.org, and a content which is the text "LyX.". In this case the root element is the p element.

The XML specification [143] does not attach any particular meaning to the element or attribute names; rather they are specific to each particular XML-based format. Our example document uses markup that has an interpretation as part of a document in XHTML [138], the XML-based reformulation of HTML. More specifically, XHTML defines that the p element contains a paragraph of text, the i element encloses content that should appear in italic font, and the a element along with its href attribute encode a Web hyperlink.

Compared to our example document, additional syntax and content besides elements and text are defined in the XML specification. Character and XML content can be referred to using *entities* of the form &*entity-name*;. Of particular importance are the entities for the characters <, >, and &, which allow the inclusion of these characters in text content without being interpreted as XML markup. As a special short-hand for elements with no content the syntax <*element-name* /> is used. To combine similar tag names from different domains, there is syntax that allows elements to be placed in different *name spaces* [140]. Additional content types include processing instructions (one is usually present at the beginning of the document to indicate XML version and the document character set), comments, and the document type declaration.

The hierarchical enclosure of elements and text with other elements forms the *document tree*, which is an ordered tree with the root element as its root. Thus, each element corresponds to a *subtree* of the document tree. The notion of a *well-formed* XML document essentially guarantees an unambiguous interpretation of the document as a tree structure, and includes such requirements as the unique root element, as well as a nesting of tags that does not violate the hierarchical containment principle. An XML document may also be associated with some *schema* [143, 142], which states further constraints beyond well-formedness. In case the XML document conforms to the rules of its schema, it is said to be *valid*.

However, the XML specification makes no reference to a tree structure, and well-formedness is not defined using a tree model. Instead XML is defined in terms of a string of Unicode [129] characters. This is also true for the *canonical* representation of an XML document [137], where the document is serialized as characters so that XML documents that would be considered equivalent will have identical canonical representations. The actual tree structure of the XML document has been accounted for in the XML Information Set [141] and the XPath data model [145] World Wide Web Consortium (W3C) recommendations.

4.2 Differencing

As the inherent syntactic structure of XML data is that of a tree, XML differencing may be approached by considering the more generic problem of differencing trees. This problem is known as the *tree-to-tree correction problem*, where a given set of tree operations and associated costs are used to find the minimum-cost sequence of operations transforming one tree to another. The cost is known as the *edit distance*. The groundwork on the tree-to-tree correction problem was laid by [118], with improvements in computational complexity by [65] and [111].

Trees may be classified into those that are *ordered*, where the order among sibling nodes matters, and those that are *unordered*, where order does not matter. Disregarding application semantics, XML itself is ordered (the order of elements matters), and therefore an ordered tree and differencing model appears more suited for the general case. XML-based formats with an ordered model include office documents [81], drawings [139], and XHTML documents, to mention a few. An unordered model may be more appropriate for, e.g., databases [130]. The unordered differencing problem has been studied in [16] and [130].

A concern with ordered tree differencing is that generalized minimum-cost tree diffing algorithms exhibit worst-case complexities from quadratic upwards [18, 101]. The quadratic upper bound of the general approach has been improved by introducing restrictions regarding the operations used and on the structure of the data [147, 17, 15].

For instance, in the algorithm by Chawathe et al. [17], assumptions on allowable parent-child relationships are made, which lowers complexity to $O\left(ne + e^2\right)$, where e is an edit distance between the inputs and n is the input size. The algorithm consists of two phases: building a matching between the input documents, and finding the minimum-cost edit script corresponding to the matching. This algorithm is used in the Open-Source `xmldiff`[1] and `diffxml`[2] XML differencing tools.

Another approach to reduce complexity is to relax the strict requirement for minimum edit distance, and replace it with greedy and heuristic approaches [16, 18]. The `xydiff` [18] tool computes the diff using a heuristic algorithm that has a computational complexity of $O(n \log n)$, which makes it attractive for use on larger documents. The designers of the XML versioning API for office documents [101] found `xydiff` to be the best tool on which to base their design.

While approaches not based on a minimum edit distance have traditionally been seen as "second-class" approaches, we find that this point can be argued [18]: An optimal diff is only optimal with respect to some cost

[1] `http://www.logilab.org/projects/xmldiff/`
[2] `http://diffxml.sourceforge.net/`

Revision 1

```
Styles
<i>Italic</i>
<pre>Typewriter</pre>
<b>Bold</b>
```

Revision 2

```
Styles in alphabetic order
<b>Emphasized</b>
<i>Italic</i>
<pre>Typewriter</pre>
```

Revision 0

```
Styles
<b>Bold</b>
<i>Italic</i>
<pre>Typewriter</pre>
```

Figure 4.2: Example XML reconciliation task

model (i.e., set of edit operations). As several models may be constructed, it is not obvious which model corresponds to the "correct" diff.

The expressiveness of the change operations computed by differencing algorithms range from those that detect inserts and deletes to those that also detect moves and copying of subtrees. Typically node insert, delete, and update operations are detected. The subtree *move* operation is a useful addition, which allows the diff to more accurately model XML restructuring, such as item reordering, rich text restructuring, or directory tree manipulations. Including the move as a primitive in tree differencing has been motivated in [18, 16, 17, 101].

4.3 Merging

As with XML differencing, XML merging, too, can be considered a special case of the corresponding computation on an abstract tree structure. However, whereas a differencing tool is able to compute changes between any well-formed inputs, with merging there is the possibility of failure due to conflicting changes. To avoid unnecessary conflicts this typically means that the merge tool needs to be aware of the semantics of the data, i.e., particularities of the XML format. Hence the problem of XML merging cannot be viewed as *only* a tree merge problem. For instance, one would usually not consider attribute ordering significant in XML, whereas such behavior is not motivated in the general tree case.

Occasionally, conflicts may only be resolved by considering application-specific semantics, and sometimes by doing this on more than one level of abstraction. Furthermore, some conflicts may go unnoticed without considering application semantics. For instance, a common assumption when merging data in a text-based format is that concurrent changes directed at different parts of the text (e.g., different lines) can be applied to the same

Revision 3

```
Styles in alphabetic order
<i>Italic</i>
<pre>Typewriter</pre>
<b>Emphasized</b>
```

Figure 4.3: Merge of revisions 1 and 2 in Figure 4.2

state, while still yielding a semantically correct result. This assumption is a large improvement over considering concurrent changes mutually exclusive, and is pervasively used.

However, this simple assumption does not guarantee correctness in all cases. For instance, consider two concurrent source code changes that both insert the declaration of a new symbol x, but at different locations. Here the assumption yields a result where the declaration of x occurs twice, while the programming language may not allow multiple declarations of the same symbol. Detecting that conflict requires considering a higher level of abstraction, where we take into account the semantics of the programming language.

It appears that to never produce an invalid result a generic XML merger would need to have full semantic knowledge of all XML-based formats. As this is quite infeasible, we find that there is no practical definition for an infallible generic XML merge procedure. Nevertheless, this does not mean that there cannot exist highly useful XML mergers, as evidenced by the text-based `diff` and `patch` tools. While these do not guarantee even syntactic correctness when used to merge XML data, they produce the desired result in a large number of cases.

A merging example for an XHTML-like document is shown in Figure 4.2, where revisions 1 and 2 are to be merged. We note that inspection of only these revisions does not yield enough information to merge. For instance, is the correct text inside the <b> element *Emphasized* or *Bold*? Thus, without auxiliary data, the two-way merge approach can only yield a conflict in this case. While there are two-way mergers for XML, such as [43], we do not consider these further here for this reason.

If we instead consider revisions 1 and 2 as derived from the base revision 0 (also shown in the Figure), we may deduce that in revision 1, the <b> subtree has been moved to the end of the list, whereas in revision 2, the word *Bold* has been changed to *Emphasized,* and the title of the list has been updated to *Styles in alphabetical order.* We find that it is indeed possible to construct a revision 3, depicted in Figure 4.3, which contains all changes. Note that in order to generate the merged revision we need to be able to reason about moved document subtrees. In a model with inserts, deletes, and updates only, we would conclude that the <b> subtree was deleted

(and a similar subtree later inserted) in revision 1 whereas it was updated in revision 2, which typically would yield a conflict.[1]

Closer inspection of the merged document exposes a problem, however. While the heading states that the list is in alphabetical order, it clearly is not. This is despite the fact that both revisions 1 and 2 are consistent with their heading. This is again an example of how data semantics affects what constitutes a valid merge. The hardness of this problem is illustrated by the fact that in this case the merger would have to understand English to detect the conflict!

In diff/patch-based merging, we merge by applying changes in the form of a diff from other replicas to an object. The diff is computed by differencing the two states S_i and S_j between which the changes occurred, and thus it encodes the edits made on either branch of the object history. To illustrate, in the above case we could either compute the diff between revision 0 and 1, and patch it to revision 2, or we could patch the diff between revisions 0 and 2 to revision 1. We note that diffs may also be readily available, e.g., in a revision repository, or in the form of edit scripts in edit-based systems.

In order for diff/patch-based merging to work the changes in the diff need to be applicable to a state different from S_i, as is pointed out in [100]. The central problem is how to identify the parts of the document to change in a manner that is both robust to deviations from S_i, and yet specific enough to address the same node that changed in S_j. For instance, if a change is encoded as "update the fifth child of the document root", the change will obviously address the wrong node if the node that was the fifth child of the root in S_i has been shifted to appear as the sixth child of the root in the state that gets patched.

One method to address this is to use unique identifiers for the nodes of the XML document that are consistent across revisions, as is done in [18]. Another approach is to add *context* to a change, such as the parent and sibling nodes of the changed node. The target document is then scanned for this context, and the change applied where the context is found. This is similar to how text files can be merged by using the context differencing facilities of `diff` and `patch`.

Context-based XML differencing is used in [100] to implement merge of XML documents as used by OpenOffice. Diff/patch-based XML merge is further used in the Xmiddle [69] middleware, where changes are generated using the XML diff tool [42].

In three-way merging we include the base state S_b in the merge process to enable additional reasoning about the changes. Furthermore, with three-way merging all three states can be kept in working memory simul-

[1] While the issue can be avoided in this particular case by considering the `<i>` and `<pre>` subtrees inserted and deleted, the issue still remains in the general case.

taneously, which simplifies correlating nodes across the input trees, and eliminates the diff/patch merge problem of how to encode the diffs in a serializable format, i.e., a format that does not use identifiers whose scope is local to a single process only, such as memory addresses.

This is illustrated in the DeltaXML [28] tool, which is able to generate a "three-way-delta" by matching the nodes of the input trees in memory and outputting a combined document where the nodes are annotated with change information. Three-way merging is then a matter of accepting all changed nodes. The node matching is based on a general tree matching algorithm that uses element identifiers and performs longest common subsequence alignment at each level of the input trees. Another three-way merging method applicable to hierarchically structured documents is outlined in [5]. This method does not include change detection, i.e., the changes between the base and the input states need to be explicitly provided.

The Harmony synchronization framework [30] combines a three-way merging strategy with an unordered tree data model and a data "filtering" language that aids reconciliation and enables synchronization across data formats. The concrete input data structures (e.g., XML documents) are first transformed to an abstract unordered tree structure suited for synchronization by, e.g., introducing keys for data elements and filling in data that may be missing from the concrete structure. The transformation, known as a *lens*, is bi-directional, so that it can be used to transform the result back to the concrete representation, or, if using another lens, to another concrete format, for cross-format synchronization. The synchronizer for the abstract tree merges any non-conflicting changes, and takes note of any conflicts for later rounds. The use of schemas to detect conflicts is also supported.

Further work on three-way merges on tree-like structures include tools for software source code merging [70], as well as state-based directory tree merging [6]. Inserts, deletes, and updates are essentially treated in the same way in all three-way merging methods we have surveyed. The difference lies in the support for a "move" operation, and the rules for merging moves.

Chapter 5

Data Synchronization in the Mobile Environment

The design space for data synchronizers is rather large. The data synchronization overview in this thesis, although focusing on optimistic approaches and state-based synchronization, still leaves open many possibilities for managing causality, detecting and propagating updates, and performing reconciliation. Considering the characteristics of the mobile environment can help us narrow down the design. In this Chapter, we consider the design features of synchronizers that have been constructed for the mobile environment. For a bird's eye view of synchronization as part of an integrated system on a mobile device, we also briefly look at mobility middleware that include synchronization as a component.

There is consensus that the optimistic rather than the pessimistic model offers better data availability in an environment with weak connectivity and intermittent disconnections [102, 107, 37, 97]. The primary caution of the optimistic model is conflicting concurrent updates, but in practice it has been found that conflicts are rare [110], and that they are usually easy to resolve [37].

One can distinguish between two major approaches to synchronization topology: infrastructure-based *client/server* and *peer-to-peer* [37]. In the client/server model, the mobile nodes exchange updates with a designated *server* node (or set of nodes), whereas in the peer-to-peer model, any node may synchronize with any other node. The peer-to-peer approach is advantageous when passing updates through the server node would be slower or more costly than direct node-to-node communication [93]. Furthermore, should the infrastructure become unavailable, no progress is possible in the client/server model.

On the other hand, not relying on a fixed infrastructure increases the demand for object replication in order to ensure availability. This consumes additional resources and can lead to poor scalability [93, 94]. One could

even make the argument that in the end, ensuring availability is easier with a client/server model [146]. A further point in favor of the client/server approach is that clients and servers need to be parted anyway, because they will be optimized for different tasks [104]. Finally, we note that there are hybrid approaches which strike a balance between the two models [94].

Reduced power and network usage is beneficial [2], and is indeed often mentioned as a design goal [37, 146, 107]. As the bandwidth and communication costs of the network may vary, we want to choose wisely *when* to use the network. Specifically, when considering whether to propagate updates immediately or after some duration, the increased opportunity to use fast and cheap connectivity is weighted against the user's need for immediate propagation of changes. Delayed propagation has the further advantage that several updates may be batched onto a single connection, and some updates may even be optimized away. [108, 37, 104] Network usage may also be reduced with *metadata-only* synchronization [126, 37], which is a technique that can reduce network usage considerably in cases when the full content of an object is not needed, but rather a description of it.

The synchronizer should tolerate network partitions or complete network disconnection [105, 97]. Disconnections are sometimes abrupt with no preceding indication, and are in some cases related to user mobility, as when leaving the office for a location without network access. Still, the user should ideally not need to provide the system with advance notification to ensure continued operation [93]. However, in practice some systems allow the user to prepare for the disconnected period by *hoarding* required data onto the device [108, 126, 146].

When considering different alternatives for the object storage model, providing a standard file system API, or having the synchronizer interface with the existing file system comes across as the logical choice [146, 126, 88]. While a customized API offers more control and does not have to account for a sometimes ill fit between synchronization and the file system model, this is offset by the utility of providing synchronization to existing file-based applications. In particular, files managed by the synchronizer should behave as ordinary files during both read and write access [116, 97].

We typically synchronize the complete file state, as obtained during a period when no writers are active, as that state is internally consistent under normal circumstances. If we, on the other hand, propagate individual writes to a file, additional mechanisms are required to ensure that a consistent state is seen [104]. Furthermore, detecting individual writes to a file usually requires access to file and operating system internals, whereas a portable application-level approach is generally preferred [107, 37].

Against this background, designing a state-based synchronizer for the standard hierarchical file system appears reasonable. However, we note that there are applications, such as video, where partial synchronization of a file is beneficial due to the large size of the complete file. [126]

A vast amount of data is already accessible on the Web through established protocols, such as HTTP, the File Transfer Protocol (FTP) [89], etc. There is thus an incentive to make a synchronizer that is interoperable with existing Web infrastructure. Using standard protocols is a good starting point [107]. We should also consider that changing the existing server infrastructure to accommodate novel systems may be unrealistic. In particular, the server versioning model may be rather limited, e.g., the only "version number" available may be a modification timestamp based on the server's internal clock [126].

As the illusion of a single replica and single-copy serializability breaks down during periods of disconnection and weak connectivity, the user will be aware of synchronization activity. It then becomes important to convey a consistent mental model of the synchronization process to the end user, so that he can understand what the synchronizer is doing. A good synchronization model can even support users in their work. For instance, a user may prefer to work in isolation, and only initiate synchronization when a mature enough version of his changes is completed [8].

In the case of concurrent modifications that cannot be merged without user intervention, we should avoid a synchronization process where the user's immediate feedback is required for further progress. Rather, the conflict should be recorded and resolution at an arbitrary node should be allowed [92]. Care needs to be taken regarding how the conflict is represented to the user. This can be especially challenging when one is limited to use the standard file system API. In [107], using a symbolic link to a nonexistent location is suggested as a method for alerting the user of a conflict when a file is accessed.

5.1 Synchronization in Mobility Middleware

Data synchronization has been proposed as a component of mobility middleware in several cases. Here we briefly describe a selection of such systems with an eye towards how user data is modeled and synchronized.

In Aura [114], the goal was to minimize distractions arising from manually managing computing resources, such as files, printers, displays, etc. This is done by providing highly automated allocation and management of computing resources and peripherals in a dynamic and heterogeneous environment. Mobile data access is provided by a version of the Coda [108] file system that has been enhanced with data staging capabilities. As Coda normally only caches a subset of files on the client, there is a possibility for cache misses. The staging capabilities consist of a staging server that keeps read-only, encrypted copies of the full set of shared files, thus alleviating cache misses.

Xmiddle [69] is mobile computing middleware for ad-hoc networks that supports mobile application development and enables transparent sharing of XML documents across heterogeneous mobile hosts, while allowing online and off-line access to data. The middleware uses optimistic replication to support disconnected operation, and includes a generic XML reconciliation mechanism for integrating concurrent updates. Besides generic reconciliation, application-specific policies are also supported for cases that require application-specific semantic knowledge. There is support for direct synchronization between nodes hosting replicas derived from the same base object, i.e., the originating node is not required to be always available.

Fugeo Core [121] is a mobility middleware which focuses on modularity, extensibility, and interoperability through the use of standardized protocols and data formats. The middleware consists of an event system [120], and components for messaging [49], data synchronization [62], presence, and reconfigurability. Host mobility is supported by HIP [74]. XML is used throughout the system as data interchange format, and reconciliation capabilities for concurrently modified XML documents are included in the synchronization component. Besides XML, the synchronizer supports efficient synchronization of hierarchical file systems.

Lime [75] is a middleware based on tuple spaces for the mobile environment with support for shared data. A *tuple* is a vector of arbitrary values, and a *tuple space* is a collection of tuples. The tuple space in Lime supports operations to put, take, and read tuples from the Lime tuple space. Synchronization happens by having nodes combine their tuple spaces into a single space. However, a tuple in the shared space is still associated with a single location, and it is removed from the space if the location disconnects. In a way this leads to a pessimistic model, as participants need to hoard the tuples they intend to use before disconnection

One.world [35] is a system that supports application development in the pervasive computing environment. Data in One.world is represented as tuples, with nesting of tuples permitted. A replication service is provided to keep data synchronized in an optimistic manner. Reconciliation is performed with the help of edit logs, which the system captures as tuples are modified.

When comparing middleware systems to stand-alone synchronizers it appears that middleware systems more easily opt for custom data models and storage options. This may be more natural in the middleware case, as these are frequently complete application environments, and to a high degree applications are expected to be built on top of the middleware. However, recognizing that it may be that only some components of a middleware get deployed outside the academic community suggests an approach of more independent components that interface using well-established standards. This design is exemplified by the Fuego Core middleware.

Chapter 6

Contributions

The published contribution of this thesis in the area of optimistic data synchronization in the mobile environment consists of Articles I–V, which are summarized and put in a larger perspective in this Chapter. As sometimes happens, the research history did not follow a straight path from topic to topic. For the benefit of the reader, we have hence forgone presenting the articles in chronological order and instead present them by synchronization topic.

We first consider Article I, where we present a three-way merging algorithm for XML data. Then, addressing the computational limitations of the target environment, in Articles II and IV we consider a lazily evaluated tree structure ("Reftree") for XML and how it can be used with the three-way merging algorithm of Article I as well as for state-based XML synchronization in general.

The three-way merger of Article I assumes a pre-existing document matching which correlates the "same" content across the input documents. However, such a matching may not be readily available, in which case we can obtain it by using an XML differencing algorithm. XML differencing also plays a natural role in XML synchronization as part of the change detection mechanism. In Article III we present an XML differencing algorithm where detecting the minimum set of edits is traded off for the mobile environment cornerstones of efficiency and simplicity. The algorithm builds on the Reftree structure presented in Articles II and IV.

The contributions of Articles I–IV are then leveraged in Article V, where an XML-aware file synchronizer for mobile devices is presented. The file synchronizer features interoperability with existing Internet infrastructure, while conforming to the requirements of the mobile environment.

6.1 State-based XML Reconciliation

In today's working environment it is common practice to edit content in a collaborative fashion. Different authors may write different parts of a document, or a document may be sent out to reviewers that make supplementary edits and add comments. In particular, multiple copies of a file may be involved, where each copy receives differing edits, while in the end a single copy that integrates all edits is desired. That is, we want to *merge* the edits in many copies into one.

With XML, the structure of data is exposed in an interoperable manner. The widespread move towards XML-based application storage formats in the recent decade has thus made the structure of application files increasingly transparent. Herein lies an opportunity to address scenarios like the one above and construct a merging algorithm which would be of relatively large scope and which would be able to process data at a relatively high semantic level.

On the matter of whether to use an edit or state-based approach, we find that applications do not generally record change operations along with the data, and furthermore, there is little consensus among applications as to what the XML change operations should be. Our goal of interoperability with existing software thus calls for a state-based approach to XML merging.

In Article I we present a state-based merger for XML that uses the three-way merging technique. Three-way merging was chosen over other approaches as it eliminates the need for a serializable format for changes, and because of the advantages of using a common base revision over other auxiliary data. Furthermore, we have proposed that the flexibility of three-way merging makes it well suited for the mobile environment [61].

We argued in Chapter 4 that there is no single "right" way to define a three-way merge for tree-structured data, but rather that many variants are possible. To find a variant with reasonably broad applicability we analyzed a set of merging use cases from the domain of XML documents with an ordered tree model. This domain includes important applications such as Web documents (XHTML), word processing and other office documents, drawings, etc.

Based on the use cases we identify an inclusion principle where we require certain fragments of the derived documents to be included in the merged document. The fragments are taken from the areas where the derived documents changed with respect to the base document. This preservation of document fragments in the merged document is an instance of the no lost updates principle. In the Article, the XML document is modeled as an ordered tree, and the fragments are informally known as contexts. These are subsequently formalized as relations that constrain the content of nodes and allowable parent and sibling relationships between nodes.

The essence of the merging algorithm is then to construct an unambiguous, or *consistent*, merged tree from the required relations. The relations from the input trees state ambiguous content or structure for the nodes around changes, and are thus normally inconsistent. The merging algorithm obtains a consistent merged tree form the inputs by discarding relations that are considered expendable. Such relations are those that do not represent a change compared to the base document. If a consistent tree is not obtained when no further relations may be discarded a conflict has occurred. The computational complexity of the algorithm is $O(n \log n)$, and in the Article we present empirical evidence that implementations can scale in accordance with this.

An important aspect that we take into account is how to merge subtree moves. In particular, our merger allows reconciliation of a moved subtree with changes inside that subtree, including moves. This allows for merging of cases where, e.g., one editor reorganizes the overall structure of a text, and another author adds some text and proofreads. The support for moves sets our approach apart from other research, including the more recent diff/patch-based approach described in [100].

In addition to the no lost updates principle, the merge has the property of being symmetrical with respect to the derived documents, i.e., it does not matter which derived document contains which changed state. Thus, no changes are given preference over others. However, in a conflict situation it may be desirable to prefer the change from one derived document over the other.

The merge relies on the ability to correlate the "same" node across trees, i.e., it assumes a *matching* [16, 17] (also: *alignment*) of tree nodes. We do not consider how to construct the matching in the Article, as that has been studied elsewhere, and those results are directly applicable. We note that the construction of accurate matchings in an efficient manner is of large practical importance.

The merging algorithm gives rise to a taxonomy for conflicts pertaining to concurrent XML modifications. We identify the *update/update conflict*, where a tree node changes in different manners (not including position in the document), the *position/position conflict*, where the position of a tree node in the merged tree is ambiguous, and the *delete/edit conflict*, where there are changes that are no longer present in the merged tree.

The contributions of this research are a three-way merging algorithm for XML with unique support for subtree moves, a set of use cases illustrating XML merging and desired outcomes, a set of design guidelines for XML three-way merging in the domain of the Article, and a classification of conflicts. Furthermore, the algorithm has been implemented and evaluated against the use cases. Both the implementation and the use cases are available as Open Source at `http://fc-raxs.googlecode.com` and `http://tdm.berlios.de`.

The author has previously described a variant of the merging algorithm in unpublished work [60]. We have furthermore published a variant suited for plain text, with specific applications to syntactically invalid XML and HTML in [63].

6.2 Lazy Trees for Data Access and Synchronization

The rapid growth of storage on mobile devices raises the concern that we cannot utilize all the data that is stored on the device efficiently. One way to address this is to adapt the data into a form more suited for the mobile device that is more compact and which requires less processing [54]. However, this breaks interoperability to some extent, and widens the gap between mobile and fixed devices. Ideally, we want to use the same data files on fixed and mobile devices alike.

In Article IV we present the *Fuego XML Stack*, which consists of algorithms and software components that allow applications to read and write verbatim XML files in a random-access fashion, while using the processing resources efficiently. By retaining XML as the on-disk storage format we stay compatible with existing XML processing applications, minimize the effort of data import and export, and eliminate the need to temporarily store the same data twice. Furthermore, data sources that can interface through an XML document model, although not necessarily using XML natively, are also easily integrated with our components.

The central idea of this work is the use of *lazy* data structures [1], i.e., data structures that are only partially instantiated in memory according to the data access pattern of the application. We provide these to applications by means of a custom API, rather than through the file system. This is because the file read-modify-write cycle is part of the problem, so compatibility in this sense cannot be retained. Nonetheless, since data remains stored in an XML file, access through the file system remains an option, whenever that becomes feasible. Examples of this include cases where the data is moved onto a fixed device, or when energy concerns are not relevant, e.g., because the mobile device is connected to a charger.

In Article II we consider state-based file system directory tree synchronization. As storage capacity increases, so does the number of files and directories that may be stored. This leads to a situation where the directory tree state becomes unmanageable due to its size. However, there is an opportunity for lazy data structures here, as we are interested in only the *changes* to the file system since the last point of synchronization rather than the full state per se.

We may now put the Fuego XML Stack to work: by providing an XML document interface to the file system directory tree and using the Stack in

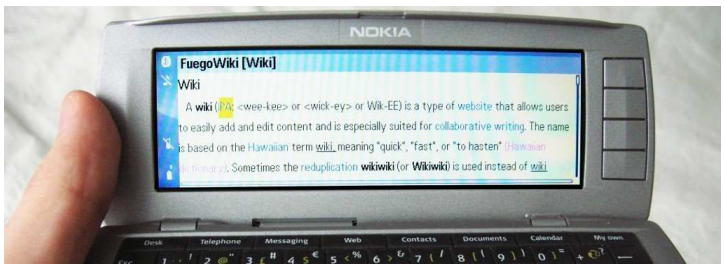

Figure 6.1: Editor for 1 GB XML file with Wikipedia content.

combination with the XML merger of Article I adapted for lazy trees, state-based reconciliation of large directory trees becomes possible.

As a further demonstration of the capabilities of the Stack we have also built an editor for a 1 GB XML file consisting of some 240 000 articles from Wikipedia[1]. The editor runs on what can nowadays be considered a modest smartphone with 4 MB RAM available per process and a 150 MHz processor. The editor, shown running on the smartphone in Figure 6.1, allows the user to look up articles by a keyword, and edit the wiki-style markup of the articles.

The Fuego XML Stack consists of three main components. The *Random Access XML Store* (RAXS) is the top-level component that provides overall XML document management such as packaging, versioning, and support for synchronization. It builds on the *XAS* component for efficient XML parsing and serialization, and the *tree-with-references* (Reftree) component for lazy XML manipulation. The terminology used for the lazy tree structures is regrettably not consistent across the Articles: they go by the name Reftrees in Article IV and *XML-with-references* (XMLR) in Article II.

RAXS is the primary API for applications that utilize the XML document random access read and write functionality. The API allows opening an XML file as a Reftree, editing that tree, and writing the changes back to the file in a transactional manner. RAXS maintains past revisions of the XML document, as well as deltas between the revisions. These are useful for three-way merging and synchronization in general, as was seen in Chapter 4. A RAXS can be thought of as a limited XML database, which is stored as a main XML file with an accompanying set of files for storing indexes, past revision, etc.

XAS provides two key pieces of functionality: out-of-order parsing of XML documents and streaming access to the unparsed bytes of an XML document during the parsing and serialization process. Out-of-order parsing in combination with an index is used in the Wikipedia editor to implement random access to articles by their title. Specifically, we do not need to parse the markup preceding the relevant article, as is required when us-

[1] `http://www.wikipedia.org`

ing a conventional XML parser. The raw byte access functionality allows copying unchanged XML content without a cycle of parsing and serialization. Furthermore, it provides access to large text content in a streaming fashion, eliminating the need for intermediary buffers. The raw byte access functionality is used in the Wikipedia Editor to manage large articles.

When accessing the XML content one uses Reftrees. In a Reftree the in-memory trees contain special nodes, known as *reference nodes*.[1] The reference nodes act as placeholders for subtrees and nodes from another tree, known as the *referenced tree*. Lazy access to a tree T is then implemented as a Reftree referencing T, where the nodes from T that are needed are present in-memory, and the rest of the nodes in T are indirectly included through appropriate reference nodes. Reftrees are also used for trees that represent a change with respect to some original tree. In this case, the reference nodes are used to include parts of the original tree that are identical in the changed tree. This construct allows us to implement mutability of a large tree without fully instantiating that tree in memory.

The Reftree API provides a set of methods for manipulating Reftrees that we have found to be of general utility: expansion of reference nodes into the nodes they reference, providing tree mutability on top of an immutable tree, combining the changes represented by Reftrees, and reversing the direction of the change represented by a Reftree. The algorithms implementing the methods are summarized in Article IV and presented in more detail in Article II.

To fully benefit from the lazy approach in synchronization we also need a reconciliation algorithm that supports Reftrees. This is considered in Article II, where the three-way merging algorithm of Article I is extended to work with Reftrees, with directory tree synchronization as the specific case being considered. We find that the extension can be done as a preprocessing step to the original algorithm. In essence we need to expand nodes in the input Reftrees, so that each node in the input trees appears in the same form across trees, i.e., either in reference or ordinary form. We call this process Reftree *normalization*. After the input Reftrees have been normalized, merging is delegated to the original algorithm.

The modular nature of the extension to the three-way merge, in combination with the regular form of normalized trees, suggest that normalization may be used when extending other processing algorithms for Reftrees as well. The essential requirement on the algorithm to extend then becomes similar to that which makes it suitable for lazy processing in the first place, namely the ability to execute without visiting some parts of the tree structure.

In Article II we also propose a method for file system change detection that is in the spirit of the lazy tree approach. A large file system is expensive

[1] The term used in Article II is *placeholder node*

to traverse in order to find changes, which is necessary if we use standard file system operations only. Our method consists of a minor alteration to the semantics of the file system that allows change detection to focus on areas of actual change.

Specifically, we propose a change to the semantics of the modification timestamp for directories, so that it maintains the most recent time of modification of the directory itself *or* of any of the descendant entries (files and directories) of the directory. Thus, with a known last time of synchronization, we only need to scan those directories which have a more recent modification stamp for changes. Similar semantics for the directory modification timestamp are used in [21], but it is used for a different purpose.

Comparing the Fuego XML Stack to other approaches, there are some notable differences. We access the XML document from the file system, while other approaches [24, 12, 14] keep the unparsed XML in memory. Our update model supports moves which, although important [16, 18], are not commonly included. We are not aware of any work which addresses lazy XML editing as comprehensively, and have found related work for the sub-components only. In particular, the idea of selective instantiation (normalization) of lazy trees for the purpose of interoperability with existing algorithms stands out.

Comparing the directory tree synchronization mechanism to distributed file systems and file synchronizers, we propose a compromise where we do augment the semantics of the file system interface, but in a very lightweight manner. Our directory synchronization mechanism is state-based, and yet exhibits edit-based scalability as it scales with the number of changes, rather than the size of the state. Furthermore, moves of files and directories are considered to be first class operations in addition to the more common insert, delete, and update operations.

The central contribution of Article IV is the Fuego XML Stack which makes it possible to process significantly larger XML files on mobile devices than is commonly considered feasible, while remaining compatible with the XML storage format. This conclusion is supported by a multi-version editor for Wikipedia articles stored in a 1 GB XML file on the mobile device, and empirical measurements of the editor that indicate that its performance is comparable to that of other standard applications available on the device. We also present experimentation with alternate approaches, which show that the Stack besides scalability can provide advantages in terms of shorter set-up time and reduced storage space consumption.

Article II contributes algorithms for normalization and other operations on Reftrees, and shows how Reftrees can be used to implement three-way merging of XML documents that do not fit in memory in their fully expanded form. In particular, we show how this method can be applied to directory tree merging. We also show empirically how change detection on

file systems can be improved over the naive scanning approach by introducing a change of semantics to the directory modification timestamp.

An Open Source implementation of the Fuego XML Stack is available at `http://fc-raxs.googlecode.com` and `http://fc-xas.googlecode.com`. The directory tree synchronization mechanism is included in the synchronization component of the Fuego Core Middleware, of which an Open Source implementation is available at `http://fc-middleware.googlecode.com`.

6.3 Efficient XML Differencing

In Articles II and IV we obtained XML deltas for synchronization and version management by capturing tree edits through a custom change API, and computing a cumulative change in the form of a Reftree. However, requiring that applications use this API for XML manipulation in the case where there are no concerns regarding XML file size is not feasible, and it contradicts the goal of retaining interoperability. Thus, we need to look for another method for obtaining XML deltas.

With only XML states available, we need to use an XML differencing algorithm to detect the changes. Furthermore, as motivated in the previous and Chapter 4, we require that the move operation should be supported. Since the available energy on the mobile device limits the amount of processing, and traditional minimum edit-distance XML differencing tools are computationally taxing, especially when the move operation is included, we also consider heuristic solutions. This narrows down the field to the algorithms presented in [17] and [18].

However, restrictions on allowable parent-descendant relationships and other assumptions regarding the form of the input XML in [17] do not fit our more general domain. The generation of a sequential edit script from the established alignment of the input trees in [17] and [18] is also superfluous in the Reftrees model, where changes are unordered. It seems that a simpler approach based on heuristics is possible.

In Article III we present a heuristic XML differencing algorithm that has been optimized for speed and simplicity, and that meshes well with the Reftrees change model. We use an approach where the tree differencing problem is translated to the domain of sequence alignment, and then transformed back to the tree domain in order to generate a delta that is expressed as tree operations, rather than operations on sequences. An ordered tree model is used, as it is applicable to a large domain of practical applications.

As elements of the sequences we use the parse tokens produced when parsing the input documents. Thus, sequential representations of the input trees are readily available, and we only need to consider how to align the

input sequences, and how to transform the aligned sequences back to the tree domain, yielding an alignment of the input trees. The delta is then generated from the aligned trees.

Sequence alignment is a well-studied problem, and there are algorithms for computing alignments that incorporate moves [125]. While these could be considered in this case, we instead use a simple heuristic alignment algorithm. This choice is motivated by the encouraging results from practical use in other tools [67, 128], as well as by being amenable to variations with different alignment heuristics.

The transformation back to the tree domain is done so that the output is a Reftree, thus making the result immediately useful for other components of the Fuego XML Stack. The idea of the sequence-to-tree alignment transformation algorithm is straightforward: subtrees corresponding to aligned subsequences are encoded as reference nodes, and subsequences without a match as ordinary nodes. However, since the sequence alignment does not respect subtree boundaries, the algorithm becomes somewhat more involved. Both the heuristic alignment and the transformation algorithms are presented in detail in the Article.

Besides the Reftree expression of the delta, we also introduce a simple XML delta format that is in essence a serialization of the delta Reftree where some repetitive patterns have been eliminated and where a method based on XPath [135] is used for addressing nodes. The format features a parallel edit model, is quite readable by humans, and shows changes using a tree structure that reflects that of the target document. A drawback of the format is that it does not encode deleted data, although this appears easy to correct.

To validate the design, and in particular the simple greedy alignment heuristics, we evaluated a Java implementation of the differencing algorithm on a set of documents from its intended domain of use. Evaluation was done by measuring the performance, both in terms of scalability and absolute quantities, and the size of the resulting deltas. Furthermore, to relate the algorithm to previous work, we also performed the measurements on similar existing XML diff tools. We also included a binary differencing tool as baseline against which to compare the tree-based approaches.

In the experiments on execution time and scalability our tool overall matched the best XML differencing approaches, which suggests that our algorithm is viable. We also observed that all XML diff tools were outperformed in terms of both execution time and compactness by the binary differencing approach.

The principal contribution of Article III is a high-performance XML differencing approach which supports move operations, and that consists of a method to map the problem to the domain of sequence alignment, an algorithm for heuristic alignment of sequences, and an algorithm for transforming aligned XML sequences to a Reftree that encodes the changes between

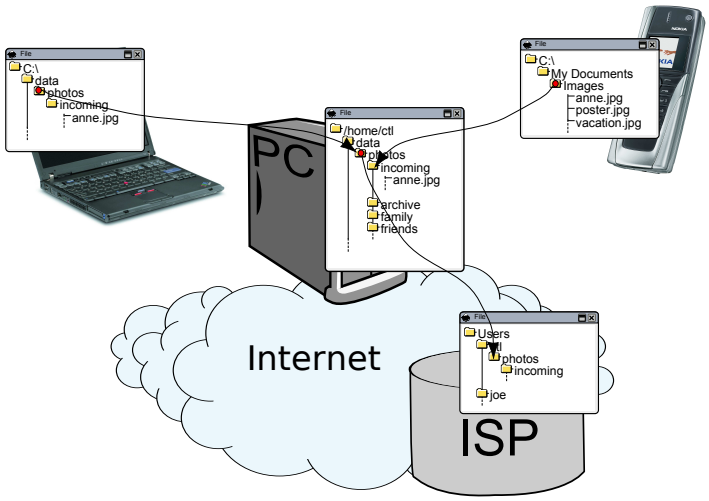

Figure 6.2: Synchronization links

the input documents. Furthermore, we contribute an implementation and measurements of its performance, scalability, and output size, as well as a quantitative comparison with existing work. We also observed that binary diffs may work very well if XML-level changes are not needed. The implementation, known as the Fuego Core XML differencing tool, is available as Open Source at `http://fc-xmldiff.googlecode.com`.

6.4 XML-aware Synchronization for Mobile Devices

The unifying theme in Articles I–IV is state-based synchronization of XML files stored on an ordinary file system. In Article V we then present a file system synchronizer with support for XML file synchronization in particular. This synchronizer, named "Syxaw", is the synchronization component of the Fuego Mobility middleware [121]. Besides integrating the work in Articles I–IV, Syxaw proposes a model for synchronization that is designed for interoperability with data sources on the Internet, and which is well suited for the mobile environment. We also consider the synchronization model easy to understand for end-users and developers alike.

Despite advances in the state of the art, data synchronization is still frequently performed manually and on an ad-hoc basis. We believe one reason for this is that existing systems do not fit well into the user environment in terms of synchronization concepts or interoperability with existing data sources. This is why we prioritized interoperability in the design, and why we think that the synchronization model should be based on simple concepts that the user ideally is already familiar with, and which accurately model the constraints of the environment. Interestingly, our analysis of

these requirements suggested that the model should be rather conservative, perhaps even old-fashioned, compared to modern systems.

The data sharing model in Syxaw is known as the "linked objects" model. In this model files and directories can have a *synchronization link* that states a synchronization target data source, typically in the form of a remote file. When synchronization is invoked on a linked file, its content is synchronized with its target object. In case of directories, the target is another directory tree, and synchronization means that the linked directory trees are synchronized.

An example scenario is shown in Figure 6.2, where synchronization links are set up between devices as indicated by the arrows. In this example, directories containing photos are linked to maintain a collection of photos across the user's devices. For instance, the smartphone stores new photographs in the `Images` directory, which is linked to the `incoming` directory on the home PC. Thus, to upload new photos from the phone to the home PC, the user invokes synchronization on the `Images` directory.

Synchronization links are set up to organize the objects in a tree synchronization topology, so that the link target of each object is its parent in the tree. We use a model of causality where each object conceptually has three revision numbers: the local revision (denoted v), and the last synchronization target and local revisions, denoted v_l and v_t. The local revision is incremented if, and only if, there is a change to the object state, and the synchronization revisions v_l and v_t are used to record the local version numbers of the object and its link target on completion of a synchronization run.

When synchronizing, v is compared to v_l and the revision v' of the link target is compared to v_t. The four possible outcomes accurately identify the cases where there were no changes ($v = v_l$, $v' = v_t$), only local changes ($v > v_l$, $v' = v_t$), only changes to the link target ($v = v_l$, $v' > v_t$), or concurrent changes of both objects ($v > v_l$, $v' > v_t$). The algorithm does not differentiate between object changes that are of local origin and those that are applied to an object in its role as a link target during synchronization. Thus, the synchronization procedure is uniform across all levels of the tree.

Figure 6.3 shows a simplified overview of the Syxaw architecture, and also how the contributions of the previous Articles fit into the architecture. At the center of Syxaw, we have the synchronization engine, which performs synchronization of linked objects, i.e., files and directories. To the operating system, Syxaw appears like any other application that reads and writes data stored on the file system. This design is portable and avoids the intricacies of development at kernel level.

Between the synchronization engine and the file system there is an abstraction for stored, synchronizable objects, known as the *object provider*. The object provider is used to provide type-specific aspects of synchronization. Syxaw supports type-specific behavior for an object's update proce-

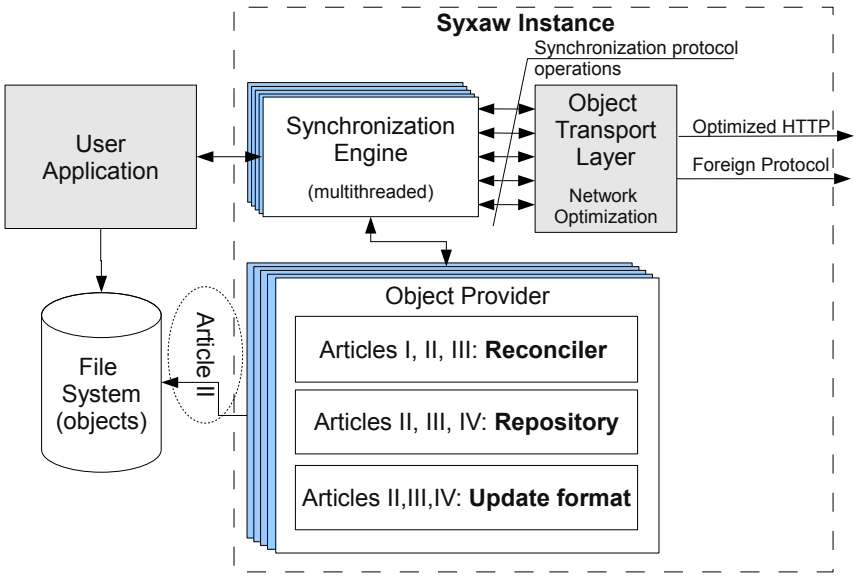

Figure 6.3: Overview of Syxaw and contributions of Articles I–IV

dure, revision history, and reconciliation, among others. For generic XML data there is an object provider where the XML differencing algorithm or the RAXS functionality is used to obtain and update objects with XML deltas, where reconciliation is based on the XML three-way merger of Article I, and where a repository of past revisions is maintained using RAXS functionality. Reftree lazy tree structures are used where applicable. The change detector presented in Article II can be used to improve file system change detection performance.

A distinguishing feature of Syxaw is that directory trees are processed as any other data objects throughout the synchronization process. They are encoded as XML documents, and rely on the generic XML object provider for basic versioning and update propagation. Reconciliation and update application, i.e., translating changes in the XML directory tree to file system operations, are unique to the directory tree type. Directory tree reconciliation is, however, not built from scratch, but rather as an augmentation of the generic XML three-way merger.

The synchronization engine communicates with remote instances either through the optimized HTTP-based Syxaw synchronization protocol, or through some other protocol as required by external data sources. For ordinary Web resources this protocol is typically HTTP, but with application-specific request formats.

The abstract synchronization protocol is designed to be very simple. The only operation that an external data source must support is the `GET` operation, which retrieves the current content of a resource. This is sufficient

for one-way synchronization. Overall, we assume only a small set of commonly available functionality on the link target node in order to be able to easily interface with existing services on the Internet.

The Syxaw synchronization protocol is a concrete serialization of the abstract synchronization protocol, and it is used when the link target, too, is a Syxaw instance. With the Syxaw protocol we optimize network usage by reducing the required bandwidth and maximally utilizing the available bandwidth (as motivated in Chapter 5). Bandwidth reduction is mainly achieved by use of compression, which may be both generic and type-specific. High bandwidth utilization is achieved by batching object requests, and using a multithreaded process flow when synchronizing objects. The use of concurrency allows, e.g., the reconciliation of one object to progress in parallel with transfer of another object.

A relatively large part of the Article is devoted to evaluating Syxaw. We qualitatively evaluate Syxaw as a provider of synchronization services to applications, and perform measurements on the efficiency of the Syxaw synchronization protocol. We also measured overall synchronization performance and resource usage for a set of realistic use cases. All evaluation was done on a smartphone connected to an authentic cellular network.

Based on the evaluation we conclude that Syxaw is a feasible approach for providing synchronization for existing as well as novel applications. Furthermore, we find that the techniques of operation batching and concurrent use of network downlink and uplink improve network utilization, but that achieving full bandwidth usage with a resource-constrained mobile client is challenging in practice. The resource usage profile of Syxaw is found to be consistent with its target environment.

Compared to other synchronization proposals, Syxaw distinguishes itself in that it interoperates transparently with resources on the World Wide Web, and in that it addresses an extensive set of synchronization-related functionality in a reasonable default manner, all while preserving the option for applications to customize any part of this functionality. In particular, Syxaw stands out in its support for reconciliation of XML data, customizable update formats, and batching and concurrent network transfer strategies.

In Article V we contribute a data synchronization model suited for interoperable synchronization on mobile devices, a comprehensive synchronization architecture based on this model, and a qualitative evaluation of the architecture as provider of synchronization functionality. We provide measurements on the effectiveness of the batching and concurrency techniques used to improve synchronization performance. Furthermore, we present findings that Web interoperability suggests that the data share model be kept simple and conservative, and that moving functionality onto the client is advantageous. An Open Source implementation of Syxaw is available at `http://fc-syxaw.googlecode.com`

Chapter 7

Discussion and Conclusions

In this thesis we set out to address data synchronization on limited mobile devices. We focused on interoperability with existing applications and Internet infrastructure, as we think this is a key enabler for ubiquitous data synchronization. As background, we characterized the mobile environment, and surveyed the topic of data synchronization with an eye towards state-based optimistic synchronization for XML data and opaque files. Three main components can be identified in optimistic data synchronization, namely update detection, propagation, and reconciliation, and we considered each of these in turn. We then presented a complete data synchronization approach, and proposed the use of lazy XML structures as an overall strategy to improve scalability.

We may now return to the research question presented in the beginning of the thesis, and consider the research presented here in light of that question, which is restated here for convenience:

> *How can we provide XML-aware file synchronization in a manner that is compatible with existing applications and suited for the mobile environment?*

As answers to this question, we can summarize our findings as follows:

- The networking environment and interoperability with existing applications suggested that synchronization should be based on an optimistic and state-based model.
- The merging aspect of synchronization was addressed by identifying patterns for merging XML, and implementing these as an XML three-way merger in Article I.
- To overcome scalability issues in processing XML data during synchronization, in Articles II and IV we proposed an approach where lazily evaluated XML documents are used.

- To address the change detection aspect, for the purposes of both merging and update propagation, either an approach based on lazily evaluated XML documents or XML differencing may be used. For XML differencing, we proposed in Article III that a relatively lightweight approach based on heuristics for aligning a sequential representation of the input documents may be used.

- The interoperability requirement affects the design of the synchronization model and protocol. In Article V, we found that a client/server model with a set of synchronization protocol operations that have semantic counterparts in popular Internet protocols supports interoperability. We also observed an incentive to move functionality onto the mobile client.

- Overall, the implementation of the research results provides a running system that demonstrates how to perform XML-aware data synchronization on mobile devices.

There has been some concerns on the performance of XML, both in the general case [78] and when considering applications on weak devices specifically [124, 52]. This has lead to the proposal of "binary" XML, i.e., serializations of the abstract XML structure that are more compact and that require less processing than XML. WAP Binary XML [134] is an early example. The W3C has a working group on the Efficient XML Interchange (EXI) Format[1], which aims to standardize one such serialization. Weak and mobile devices are targeted by this work.

Throughout the research carried out here, standard XML has been used because of the interoperability advantages that it provides, and for the reason of interoperability we have explicitly avoided transcoding XML data into an alternate format. Given that we, too, address performance issues when using XML, some comparative observations with alternate serializations can be made.

Our results help demarcate the boundary between when XML and when an alternate serialization should be used. Specifically, in cases where only limited parts of the document need to be accessed it seems beneficial to use XML, as reasonable performance can be obtained with the use of lazy XML structures, while XML compatibility is retained. Furthermore, as one could conceivably implement lazy tree structures similar to those presented here for an alternate format, the approaches appear complementary rather than exclusive.

Because of the limited resources of mobile devices, it is commonly argued that processing should be offloaded to nodes in the fixed network. In

[1] `http://www.w3.org/TR/exi/`

contrast to this, in constructing the data synchronizer we found that moving some functionality onto the mobile device brings advantages in terms of deployability on the Internet. In particular, we advocate an approach where merging of concurrent changes happen on the mobile device, rather than the fixed node. However, this approach emphasizes the need for efficient algorithms.

Decentralized systems for dissemination of data where each node is roughly equal in terms of capabilities (so-called peer-to-peer systems) are currently quite popular. The idea of addressing scalability, availability, and fault tolerance issues by increasing the number of nodes in a distributed system is indeed appealing. In contrast to this trend, our approach is a more traditional client/server design. We find that this model allows us to naturally interact with the existing Internet server infrastructure, and that the client/server approach lends itself to a rather straightforward model of causality based on simple integer revision numbers. Furthermore, as pointed out in Chapter 5, with a client/server approach we can reduce network usage. The obvious drawback of the approach is the reliance on centralized infrastructure.

Interoperability with the current Internet by using standard protocols is generally considered beneficial (e.g., [107]), and HTTP has become something of a generalized transport layer, on top of which new protocols are developed. However, it seems less common to consider how to choose protocol operations so that these can be mapped to operations on existing Web services, as we have done in this thesis.

While we have constructed an integrated approach for data synchronization on mobile devices, we note that some of the results are likely to be applicable outside this original context. The XML three-way merging algorithm seems suited for use in document revision management systems, and indeed it was presented at a document engineering venue. This is true for the XML differencing approach as well, which has found uses, e.g., in the W3C EXI working group for verifying candidate implementations. Finally, the lazy XML structures could be used on fixed nodes as well to enable applications to store even larger sets of data as XML, rather than have to use proprietary formats.

There are some further venues of research which we considered during this thesis work, but which were not carried out far enough to obtain results suitable for publication. Reftree-based synchronization of XML document subsets could be used when only some parts of an XML document is needed on the mobile device. The synchronization model presented here may potentially be extended to allow for peer-to-peer collaboration in cases when the Internet infrastructure is unavailable. This could potentially include epidemic dissemination of updates, while retaining a centralized node for committing updates.

Future work includes a more detailed study of the feasibility of building application-specific mergers on top of the generic XML merger presented here. The effect of matching accuracy in XML merging should also be researched, in order to determine what the implications of an "incorrect" matching may be on the merge result. Specifically, access to a large corpus of structured data that has undergone concurrent editing would enable validation and potential refinement of the three-way merger.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[2] S. Agarwal, D. Starobinski, and A. Trachtenberg. On the scalability of data synchronization protocols for PDAs and mobile devices. *IEEE Network*, 16(4):22–28, 2002.

[3] P. S. Almeida, C. Baquero, and V. Fonte. Version stamps – decentralized version vectors. In *ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 544–544. IEEE Computer Society, July 2002.

[4] A. Anand, C. Manikopoulos, Q. Jones, and C. Borcea. A quantitative analysis of power consumption for location-aware applications on smart phones. In *Proceedings of the 2007 IEEE International Symposium on Industrial Electronics*, pages 1986–1991, Vigo, Spain, June 2007.

[5] U. Asklund. Identifying conflicts during structural merge. In *Proceedings of the Nordic Workshop on Programming Environment Research*, pages 231–242, Lund, Sweden, June 1994.

[6] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 98–108, Oct. 1998.

[7] K. C. Barr and K. Asanovic. Energy-aware lossless data compression. *ACM Transactions on Computer Systems*, 24(3):250–291, Aug. 2006.

[8] M. Bento and N. Preguica. Operational transformation based reconciliation in the FEW file system. In *IWCES8: Proceedings of the Eight International Workshop on Collaborative Editing Systems*, IEEE Distributed Systems Online. Institute of Electrical and Electronic Engineers, Nov. 2006.

[9] T. Berners-Lee, R. T. Fielding, and L. Masinter. *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. Internet Engineering Task Force, Jan. 2005.

[10] Bluetooth SIG. *Specification of the Bluetooth System, Core Package version 2.0*, Nov. 2004.

[11] J. Border, M. Kojo, et al. *RFC 3135: Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations*. Internet Engineering Task Force, June 2001.

[12] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In J. C. Freytag, P. C. Lockemann, et al., editors, *29th International Conference on Very Large Data Bases*, pages 141–152. Morgan Kaufmann Publishers, Sept. 2003.

[13] J. Cai and D. J. Goodman. General packet radio service in GSM. *IEEE Communications Magazine*, 35(10):122–131, Oct. 1997.

[14] B. Catania, B. C. Ooi, W. Wang, and X. Wang. Lazy XML updates: Laziness as a virtue of update and structural join efficiency. In F. Özcan, editor, *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 515–526, June 2005.

[15] S. S. Chawathe. Comparing hierarchical data in external memory. In M. P. Atkinson, M. E. Orlowska, et al., editors, *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 90–101, San Francisco, CA, USA, Sept. 1999. Morgan Kaufmann Publishers Inc.

[16] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In J. Peckham, editor, *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 26–37. ACM Press, May 1997.

[17] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, June 1996.

[18] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *18th International Conference on Data Engineering*, pages 41–52, Feb. 2002.

[19] B. Cohen. Incentives build robustness in BitTorrent. In *1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003. Proceedings online.

[20] G. Colouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, Boston, Massachusetts, USA, 2nd edition, 1994.

[21] L. P. Cox and B. D. Noble. Fast reconciliations in fluid replication. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, pages 449–458. IEEE Computer Society, Apr. 2001.

[22] A. Demers, D. Greene, et al. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[23] R. Dettmer. GSM: European cellular goes digital. *IEE Review*, 37(7):253–257, July 1991.

[24] R. Fernandes and M. Raghavachari. Inflatable XML processing. In G. Alonso, editor, *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference*, volume 3790 of *Lecture Notes in Computer Science*, pages 144–163, Heidelberg, Germany, Nov. 2005. Springer-Verlag.

[25] R. Fielding, J. Gettys, et al. *RFC 2616: Hypertext Transfer Protocol — HTTP/1.1*. Internet Engineering Task Force, June 1999.

[26] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[27] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 70–75, Mar. 1982.

[28] R. L. Fontaine. Merging XML files: a new approach providing intelligent merge of XML data sets. In *Proceedings of XML Europe 2002*, May 2002.

[29] J. N. Foster, M. B. Greenwald, et al. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246. ACM Press, 2005.

[30] J. N. Foster, M. B. Greenwald, et al. Exploiting schemas in data synchronization. *Journal of Computer System Sciences*, 73(4):669–689, 2007.

[31] B. Gates. *2008 International Consumer Electronics Show Keynote*, Jan. 2008. Transcript available online.

[32] D. Gavalas and D. Economou. The technology landscape of wireless web. *International Journal of Mobile Communications*, 5(5):508–527, 2007.

[33] Google Inc., Mountain View, California, USA. *Android Operating System Source Code 1.0*, 2008.

[34] J. Gray. A conversation with Tim Bray. *ACM Queue*, 3(1):20–25, Feb. 2005.

[35] R. Grimm, J. Davis, et al. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4):421–486, Nov. 2004.

[36] R. Guy, G. Popek, and T. Page, Jr. Consistency algorithms for optimistic replication. In *International Conference on Network Protocols*, pages 250–261, San Francisco, CA, USA, 1993. Institute of Electrical and Electronic Engineers.

[37] R. Guy, P. Reiher, et al. Rumor: Mobile data access through optimistic peer-to-peer replication. In *17th International Conference on Conceptual Modeling (ER98): Workshop on Mobile Data Access*, pages 21–24, Nov. 1998.

[38] A. Helal, A. Khushraj, and J. Zhang. Incremental hoarding and reintegration in mobile environments. In *Proceedings of the 2002 Symposium on Applications and the Internet*, pages 8–11, Feb. 2002.

[39] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[40] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global positioning system: theory and practice*. Springer-Verlag, New York; Wien, 1992.

[41] HTC Corp., Taiwan, R. O. C. *HTC Dream Specification*, 2009.

[42] IBM Alphaworks. *XML TreeDiff*, 1998.

[43] IBM Alphaworks. *XML Diff and Merge Tool*, 1999.

[44] Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA. *IEEE Std 802.11 — Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, Mar. 1999.

[45] International Organization for Standardization, Geneva, Switzerland. *ISO 8879:1986. Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*, 1986.

[46] International Organization for Standardization, Geneva, Switzerland. *JPEG Standard ISO/IEC 10918-1/ITU-T Recommendation T.81*, 1993.

[47] D. B. Johnson, C. E. Perkins, and J. Arkko. *RFC 3775: Mobility Support in IPv6*. Internet Engineering Task Force, June 2004.

[48] B. B. Kang, R. Wilensky, and J. Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 670–670. IEEE Computer Society, May 2003.

[49] J. Kangasharju. *XML Messaging for Mobile Devices*. PhD thesis, University of Helsinki, Department of Computer Science, Helsinki, Finland, Jan. 2008.

[50] J. Kangasharju, T. Lindholm, et al. Collaborative XML editing on small devices: An application of mobility middleware. In *Pervasive 2007 Demo Session*, May 2007.

[51] J. Kangasharju, T. Lindholm, and S. Tarkoma. XML security with binary XML for mobile Web services. *International Journal of Web Services Research*, 5(3):1–19, July 2008.

[52] J. Kangasharju, S. Tarkoma, and T. Lindholm. Xebu: A binary format with schema-based optimizations for XML data. In A. H. H. Ngu, M. Kitsuregawa, et al., editors, *The 6th International Conference on Web Information Systems Engineering*, volume 3806 of *Lecture Notes in Computer Science*, pages 528–535, Heidelberg, Germany, Nov. 2005. Springer-Verlag.

[53] O. Kassinen, T. Koskela, E. Harjula, and M. Ylianttila. Case study on Symbian OS programming practices in a middleware project. *Studies in Computational Intelligence*, 150:89–99, 2008.

[54] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[55] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218. ACM Press, 2001.

[56] S. Keshav. Why cell phones will dominate the future Internet. *Computer Communication Review*, 35(2):83–86, 2005.

[57] F. Lam, N. Lam, and R. Wong. Efficient synchronization for mobile XML data. In *12th ACM Conference on Information and Knowledge Management*, pages 153–160, Nov. 2004.

[58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[59] Y.-W. Lee, K.-S. Leung, and M. Satyanarayanan. Operation shipping for mobile file systems. *IEEE Transactions on Computers*, 51(12):1410–1422, Dec. 2002.

[60] T. Lindholm. A 3-way merging algorithm for synchronizing ordered trees — the 3DM merging and differencing tool for XML. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, Sept. 2001.

[61] T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Third ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 93–97, Sept. 2003.

[62] T. Lindholm, J. Kangasharju, and S. Tarkoma. Syxaw: Data synchronization middleware for the mobile web. *Mobile Networks and Applications*, 14(5):661–676, 2009.

[63] T. Lindholm and T. Rüger. A fault-tolerant three-way merge for XML and HTML. In M. H. Hamza, editor, *Proceedings of the Ninth IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 71–76. ACTA Press, Feb. 2005.

[64] Q. Lu and M. Satyanarayanan. Improving data consistency in mobile computing using isolation-only transactions. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 124–128. Institute of Electrical and Electronic Engineers, 1995.

[65] S.-Y. Lu. A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):219–224, Apr. 1979.

[66] P. Lucas. Mobile devices and mobile data — issues of identity and reference. *Human-Computer Interaction*, 16(2):323–336, 2001.

[67] J. MacDonald. File system support for delta compression. Master's thesis, UC Berkeley, May 2000.

[68] A. Marian, S. Abiteboul, G. Cobéna, and L. Mignet. Change-centric management of versions in an XML warehouse. In P. M. G. Apers, P. Atzeni, et al., editors, *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 581–590. Morgan Kaufmann Publishers Inc., Sept. 2001.

[69] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A data-sharing middleware for mobile computing. *Personal and Wireless Communications*, 21(1):77–103, Apr. 2002.

[70] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[71] R. Min, M. Bhardwaj, et al. Low-power wireless sensor networks. In *Fourteenth International Conference on VLSI Design*, pages 205–210, Swizerland, 2001. Inderscience.

[72] P. Mockapetris. *RFC 1034: Domain Names — Concepts and Facilities*. Internet Engineering Task Force, Nov. 1987.

[73] P. Mockapetris. *RFC 1035: Domain Names — Implementation and Specification*. Internet Engineering Task Force, Nov. 1987.

[74] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. *RFC 5201: Host Identity Protocol*. Internet Engineering Task Force, Apr. 2008. [Experimental].

[75] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: a coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3):279–328, 2006.

[76] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, Banff, Alberta, Canada, 2001.

[77] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[78] M. Nicola and J. John. XML parsing: a threat to database performance. In *12th ACM Conference on Information and Knowledge Management*, pages 175–178, Nov. 2003.

[79] K. J. O'Brien. Panel rejects Microsoft's open format. The New York Times, September, 5, 2007.

[80] D. O'Mahony. UMTS: the fusion of fixed and mobile networking. *IEEE Internet Computing*, 2(1):49–56, Jan. 1998.

[81] Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. *Open Document Format for Office Applications (OpenDocument) v1.1*, Feb. 2007. OASIS Standard.

[82] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *CSCW '06: Proceedings of the 2006 ACM conference on Computer supported cooperative work*, pages 259–268. ACM Press, 2006.

[83] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, Jan. 2005.

[84] D. S. Parker, Jr., G. Popek, et al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[85] P. Pavan, R. Bez, P. Olivo, and E. Zononi. Flash memory cells: An overview. *Proceedings of the IEEE*, 85(8):1248–1271, 1997.

[86] C. E. Perkins. Ad hoc networking: an introduction. In C. E. Perkins, editor, *Ad Hoc Networking*, pages 1–28. Addison-Wesley, Boston, Massachusetts, USA, 2001.

[87] K. Petersen, M. Spreitzer, et al. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, Sept. 1997.

[88] G. J. Popek, R. G. Guy, T. W. Page, Jr, and J. S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the First Workshop on the Management of Replicated Data*, pages 20–25, Huston, TX, USA, Nov. 1990. Institute of Electrical and Electronic Engineers.

[89] J. Postel and J. Reynolds. *RFC 959: File Transfer Protocol (FTP)*. Internet Engineering Task Force, Oct. 1985.

[90] K. Raatikainen, H. B. Christensen, and T. Nakajima. Application requirements for middleware for mobile and pervasive systems. *Mobile Computing and Communications Review*, 6(4):16–24, Oct. 2002.

[91] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University, 1981.

[92] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, Vienna, Austria, 2001.

[93] D. Ratner, P. Reiher, G. Popek, and G. H. Kuenning. Replication requirements in mobile environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[94] D. Ratner, P. Reiher, and G. J. Popek. Roam: a scalable replication system for mobility. *Mobile Networks and Applications*, 9(5):537–544, 2004.

[95] P. Reiher, J. Heidemann, et al. Resolving file conflicts in the Ficus file system. In *1994 USENIX Summer Conference*, pages 183–195, June 1994.

[96] S. Rhea, C. Wells, et al. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, Sept. 2001.

[97] B. Richard, D. Mac Nioclais, and D. Chalon. Clique: a transparent, peer-to-peer collaborative file sharing system. In M.-S. Chen, P. K. Chrysanthis, M. Sloman, and A. B. Zaslavsky, editors, *4th International Conference on Mobile Data Management*, volume 2574 of *Lecture Notes in Computer Science*, pages 21–24, Heidelberg, Germany, Jan. 2003. Springer-Verlag.

[98] O. Riva and J. Kangasharju. Challenges and lessons in developing middleware on smart phones. *IEEE Computer*, Oct. 2008.

[99] D. Roundy. Darcs: distributed version management in Haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, Tallinn, Estonia, Sept. 2005. ACM Press.

[100] S. Rönnau, C. Pauli, and U. M. Borghoff. Merging changes in XML documents using reliable context fingerprints. In D. C. A. Bulterman, L. F. G. Soares, and M. da Graça C. Pimentel, editors, *ACM Symposium on Document Engineering*, pages 52–61, Sao Paulo, Brazil, Sept. 2008.

[101] S. Rönnau, J. Scheffczyk, and U. M. Borghoff. Towards XML version control of office documents. In *ACM Symposium on Document Engineering*, pages 10–19. ACM Press, Nov. 2005.

[102] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[103] I. Salmre. *Writing mobile code: essential software engineering for building mobile applications*. Addison-Wesley, New York, LLC, Feb. 2005.

[104] M. Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, 18(1):1–9, 1992.

[105] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1):26–33, Feb. 1996.

[106] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug. 2001.

[107] M. Satyanarayanan. The evolution of Coda. *ACM Transactions on Computer Systems*, 20(2):85–124, 2002.

[108] M. Satyanarayanan and J. Kistler. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[109] M. Satyanarayanan, M. Kozuch, C. Helfrich, and D. R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Pervasive and Mobile Computing*, 1(2):157–189, 2005.

[110] M. Satyanaraynan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[111] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, Dec. 1977.

[112] P. Serrano-Alvarado, C. Roncancio, and M. Adiba. A survey of mobile transactions. *Distributed and Parallel Databases*, 16(2):193–230, 2004.

[113] S. Shepler, B. Callaghan, et al. *RFC 3530: Network File System (NFS) version 4 Protocol*. Internet Engineering Task Force, Apr. 2003.

[114] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Software Architecture: System Design, Development, and Maintenance (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture)*, pages 29–43. Kluwer Academic Publishers, Aug. 2002.

[115] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. ACM Press, 1998.

[116] E. Swierk, E. Kiciman, et al. The Roma personal metadata service. *Mobile Networks and Applications*, 7(5):407–418, 2002.

[117] SyncML Initiative. *SyncML Sync Protocol, version 1.1*, Feb. 2002.

[118] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.

[119] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, New Jersey, USA, 2nd edition, 2001.

[120] S. Tarkoma. Fuego toolkit: a modular framework for content-based routing. In R. Baldoni, editor, *Proceedings of the Second International Conference on Distributed Event-Based Systems, DEBS 2008*, pages 325–328, July 2008.

[121] S. Tarkoma, J. Kangasharju, T. Lindholm, and K. Raatikainen. Fuego: Experiences with mobile data communication and synchronization. In *17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Sept. 2006.

[122] D. B. Terry, A. J. Demers, et al. Session guarantees for weakly-consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Sept. 1994.

[123] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[124] M. Tian, T. Voigt, et al. Performance considerations for mobile web services. *Computer Communications*, 27(11):1097–1105, July 2004.

[125] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[126] J. Tolvanen, T. Suihko, J. Lipasti, and N. Asokan. Remote storage for mobile devices. In *First International Conference on Communication System Software and Middleware COMSWARE*, pages 1–9, Delhi, India, 2006. Institute of Electrical and Electronic Engineers.

[127] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–195, 1999.

[128] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Canberra, Australia, Feb. 1999.

[129] Unicode Consortium. *The Unicode Standard, Version 4.0*. Addison-Wesley, Boston, Massachusetts, USA, Aug. 2003.

[130] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: an effective change detection algorithm for XML documents. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *19th International Conference on Data Engineering*, pages 519–530. Institute of Electrical and Electronic Engineers, Mar. 2003.

[131] WAP Forum. *Wireless Application Protocol: Architecture Specification*, 2001.

[132] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[133] World Wide Web Consortium, Cambridge, Massachusetts, USA. *HTML 4.01 Specification*, Dec. 1999. W3C Recommendation.

[134] World Wide Web Consortium, Cambridge, Massachusetts, USA. *WAP Binary XML Content Format*, June 1999. W3C Note.

[135] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Path Language (XPath) 1.0*, Nov. 1999. W3C Recommendation.

[136] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XSL Transformations (XSLT) Version 1.0*, Nov. 1999. W3C Recommendation.

[137] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Canonical XML Version 1.0*, Mar. 2001. W3C Recommendation.

[138] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*, Aug. 2002. W3C Recommendation.

[139] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Scalable Vector Graphics (SVG) 1.1 Specification*, Jan. 2003. W3C Recommendation.

[140] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Namespaces in XML 1.1*, Feb. 2004. W3C Recommendation.

[141] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Information Set*, 2nd edition, Feb. 2004. W3C Recommendation.

[142] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Schema Part 1: Structures*, 2nd edition, Oct. 2004. W3C Recommendation.

[143] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Extensible Markup Language (XML) 1.0*, 4th edition, Aug. 2006. W3C Recommendation.

[144] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XQuery 1.0: An XML Query Language*, Jan. 2007. W3C Recommendation.

[145] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, Jan. 2007. W3C Recommendation.

[146] J. Zhang, A. Helal, and J. Hammer. Ubidata: Ubiquitous mobile file service. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 893–900, Melbourne, Florida, USA, 2003. ACM Press.

[147] K. Zhang and D. Shasha. Fast algorithm for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581–621, Dec. 1990.

List of Figures

Index

ISBN 978-952-248-212-9
ISBN 978-952-248-213-6 (PDF)
ISSN 1795-2239
ISSN 1795-4584 (PDF)